US009455955B2

(12) **United States Patent**
Fetik

(10) **Patent No.: US 9,455,955 B2**
(45) **Date of Patent: Sep. 27, 2016**

(54) **CUSTOMIZABLE STORAGE CONTROLLER WITH INTEGRATED F+ STORAGE FIREWALL PROTECTION**

(71) Applicant: **Richard Fetik**, Monterey, CA (US)

(72) Inventor: **Richard Fetik**, Monterey, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **13/925,822**

(22) Filed: **Jun. 24, 2013**

(65) **Prior Publication Data**

US 2014/0020083 A1 Jan. 16, 2014

Related U.S. Application Data

(63) Continuation-in-part of application No. 12/820,137, filed on Jun. 21, 2010, now Pat. No. 8,474,032, which is a continuation-in-part of application No. 11/803,947, filed on May 15, 2007, now Pat. No. 7,743,260.

(60) Provisional application No. 60/747,536, filed on May 17, 2006.

(51) **Int. Cl.**

G06F 11/36	(2006.01)
H04L 29/06	(2006.01)
G06F 21/55	(2013.01)
G06F 21/44	(2013.01)
G06F 21/62	(2013.01)
G06F 21/78	(2013.01)
H04L 12/22	(2006.01)

(52) **U.S. Cl.**

CPC ***H04L 63/0209*** (2013.01); ***G06F 21/44*** (2013.01); ***G06F 21/552*** (2013.01); ***G06F 21/6209*** (2013.01); ***G06F 21/78*** (2013.01); ***H04L 63/101*** (2013.01); *G06F 2221/2141* (2013.01); *H04L 63/0227* (2013.01)

(58) **Field of Classification Search**

None

See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

6,275,588 B1 *	8/2001	Videcrantz et al.	380/255
7,181,603 B2 *	2/2007	Rothrock et al.	713/1
7,260,726 B1 *	8/2007	Doe et al.	713/189

OTHER PUBLICATIONS

NASD Scalable Storage Systems. Gibson et al. Proceedings of USENIX, Linux Workshop(1999).*

Security vs Performance: Tradeoffs using a Trust Framework. Singh et al. IEEE(2005).*

Implementation of a Reconfigrable Data Protection Module for NoC-base MPSoCs. Firorin et al.IEEE(2008).*

An Apparatus realizing switch of computing device status. Wang et al. IEEE(2006).*

* cited by examiner

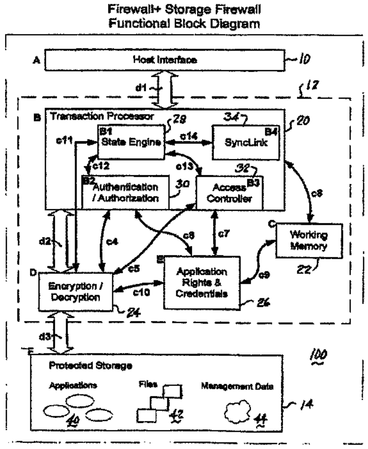
Primary Examiner — Venkat Perungavoor

(74) *Attorney, Agent, or Firm* — Hamrick IP-Law Office; Claude A. S. Hamrick

(57) **ABSTRACT**

A Customizable Storage Controller (CSC) is a software defined storage device controller, a replacement for the ASIC storage controller approach that has been used up to now. The differences from the current storage controllers are that the CSC software will need to be protected from unauthorized modification and provides an excellent place to add additional storage management functionality. The CSC type of storage controller is a good place to integrate the F+ Storage Firewall storage protection technology, fitting the needs of the CSC as well as protecting stored data from unauthorized access. This portion of the larger patent disclosure provides the design of a CSC both with a software version of a F+ Storage Firewall, as well as an improved (more secure) CSC designed with a security co-processor and locked firmware. These designs can be implemented with standard parts such as microprocessors and/or FPGAs (Field Programmable Gate Arrays), RAM (Random Access Memory), and some version of nonvolatile memory as a program store.

10 Claims, 12 Drawing Sheets



Firewall+ Storage Firewall
Functional Block Diagram

Firewall+ Storage Firewall
Functional Block Diagram

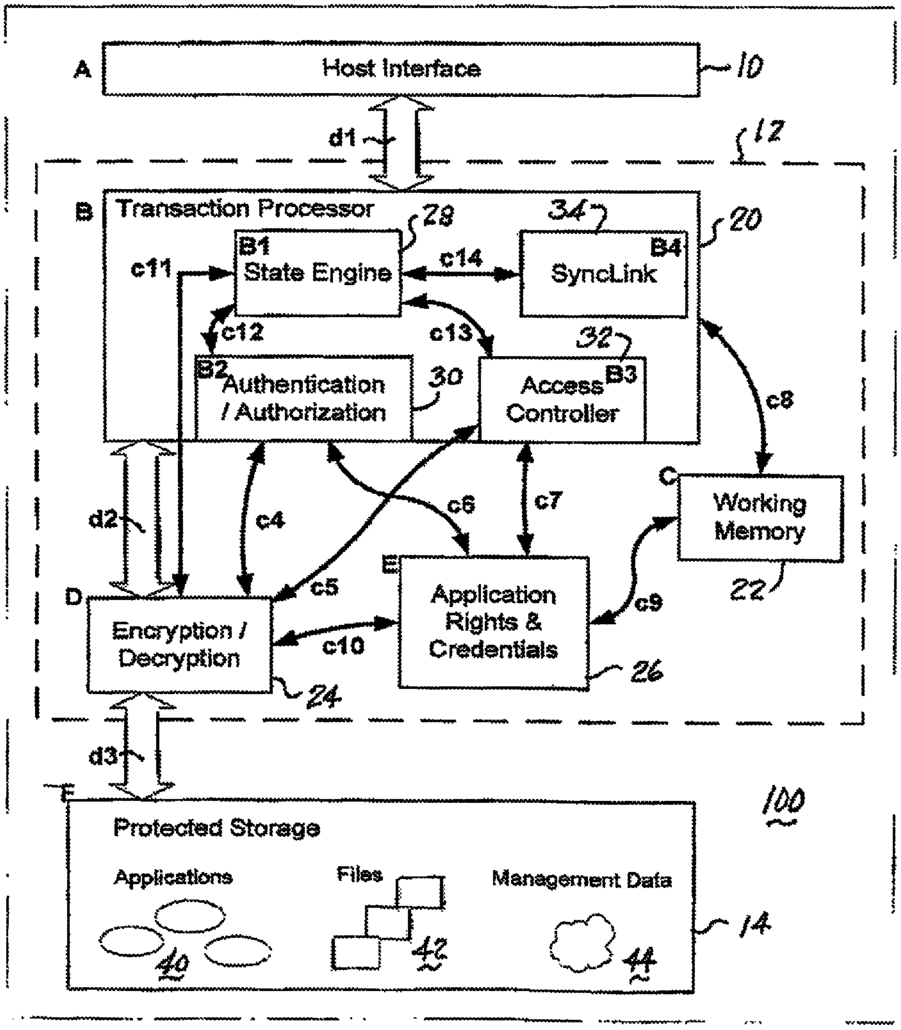


Fig. 1

Bubble Design: What's Inside

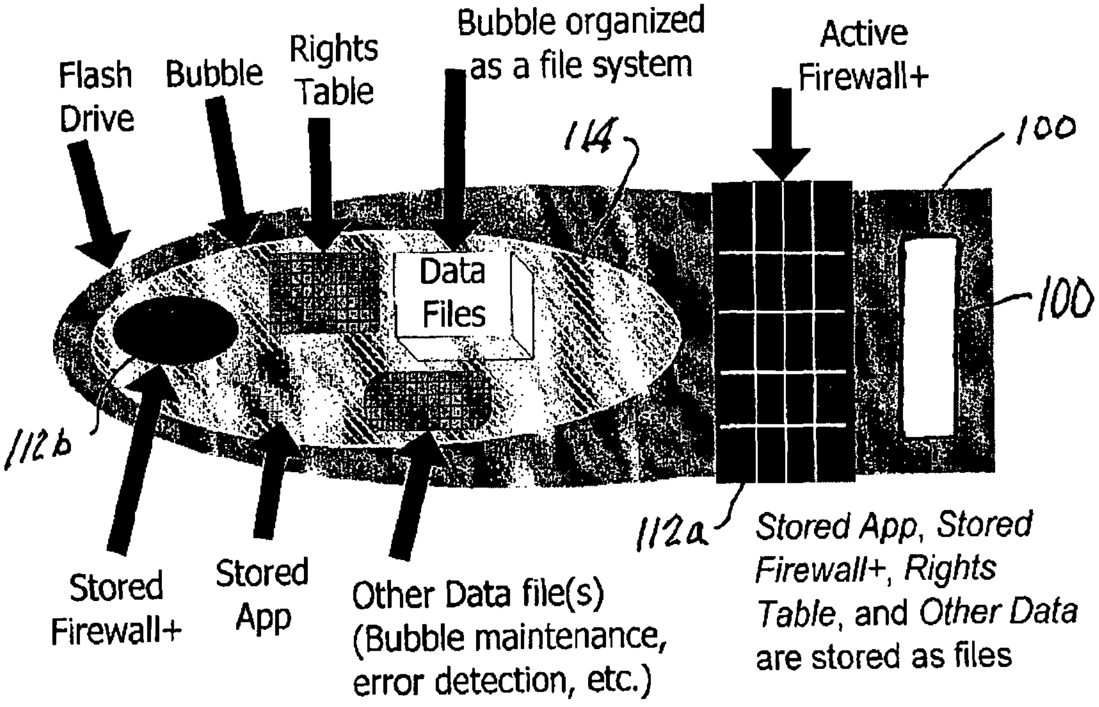

Flash Drive
Bubble
Rights Table
Bubble organized as a file system
Active Firewall+
114
100
Data Files
100
112b
112a
Stored Firewall+
Stored App
Other Data file(s) (Bubble maintenance, error detection, etc.)
Stored App, *Stored Firewall+*, *Rights Table*, and *Other Data* are stored as files

Fig. 2

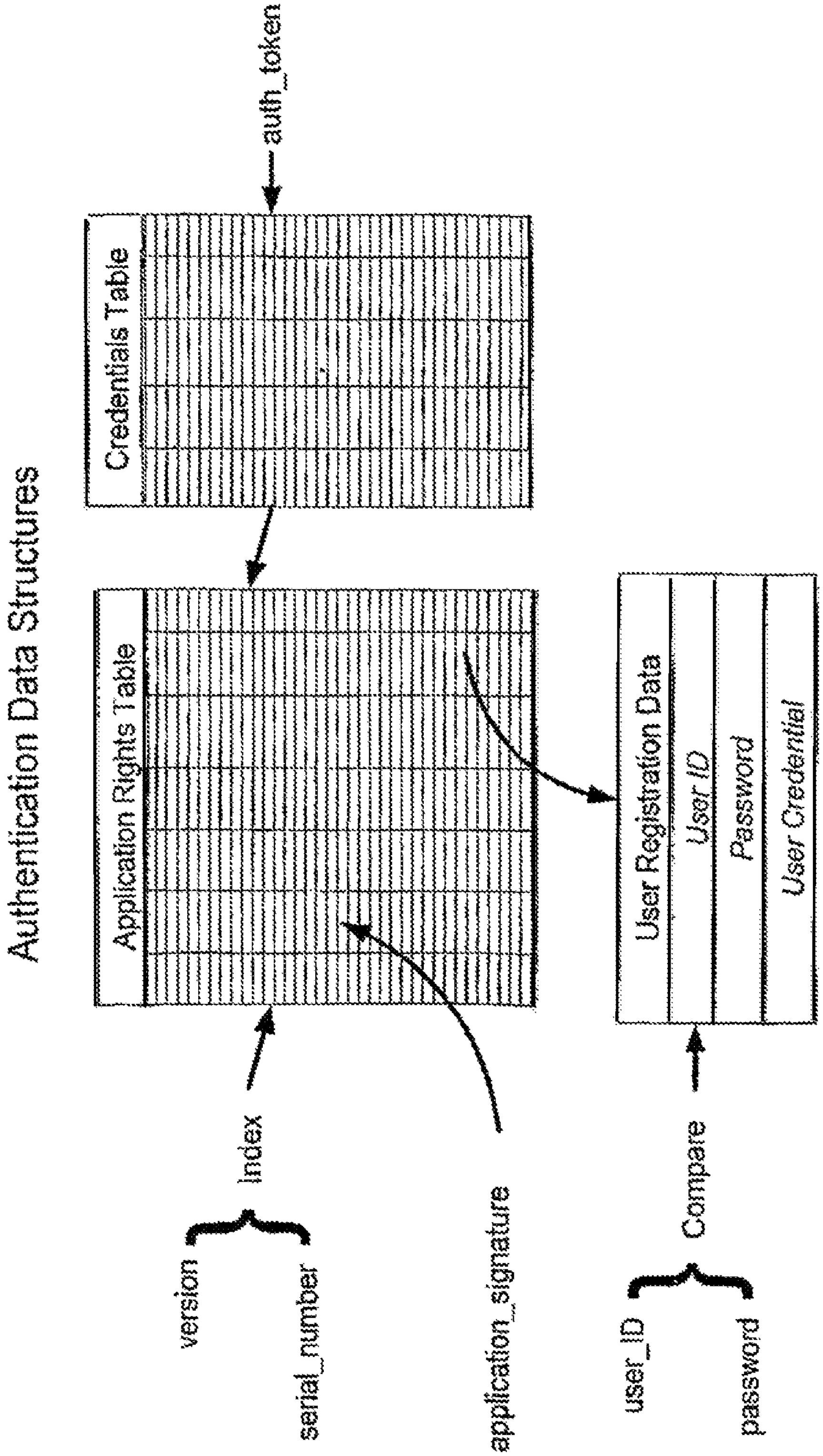
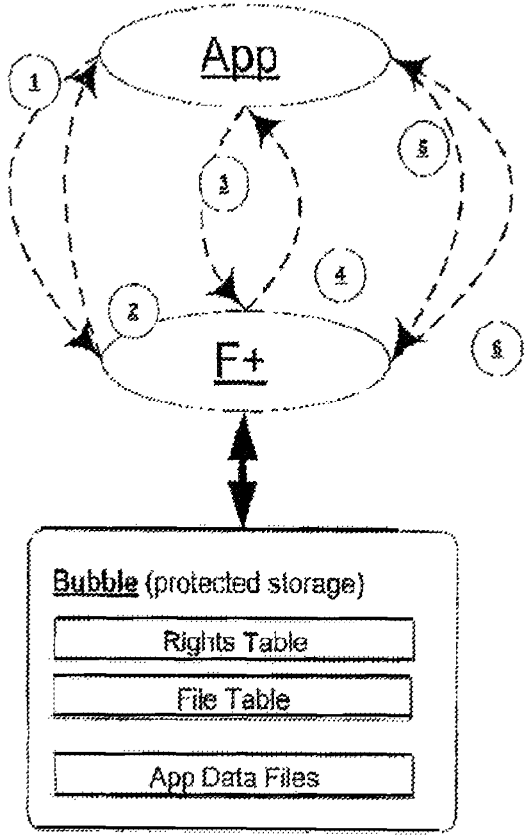

Authentication Data Structures
auth_token
Credentials Table
Application Rights Table
User Registration Data
User ID
Password
User Credential
version
serial_number
Index
application_signature
user_ID
password
Compare

Fig. 3

Interaction between Firewall+ Storage
Firewall (F+) and Application (App)

where App
- F+ API initiation request is in App startup code,
- is linked to F+ ABI, and
- App is registered in the F+ Rights table



(1) App starts to run, initiates contact with F+

(2) F+ authenticates App, provides authorization credential (auth_token), where the F+ ABI keeps track of it

(3) App sends 'open for read' file access request to F+, wher auth_token is added to the fopen() by the F+ ABI

(4) F+ (optionally logs access request), verifies App's auth_token, opens the file for read, returns a file reference – an index into the File Table

(5) App sends read access request to F+, where auth_token added to the fread() by the F+ ABI, and the file pointer (fp) is F+ file reference, an index into the File Table

(6) F+ (optionally logs access request), verifies App's auth_token, reads requested data from file, updates entry in the File Table indicating current position in the file, returns th requested data to the App

Fig. 4

Firewall+ System Functional Block Diagram

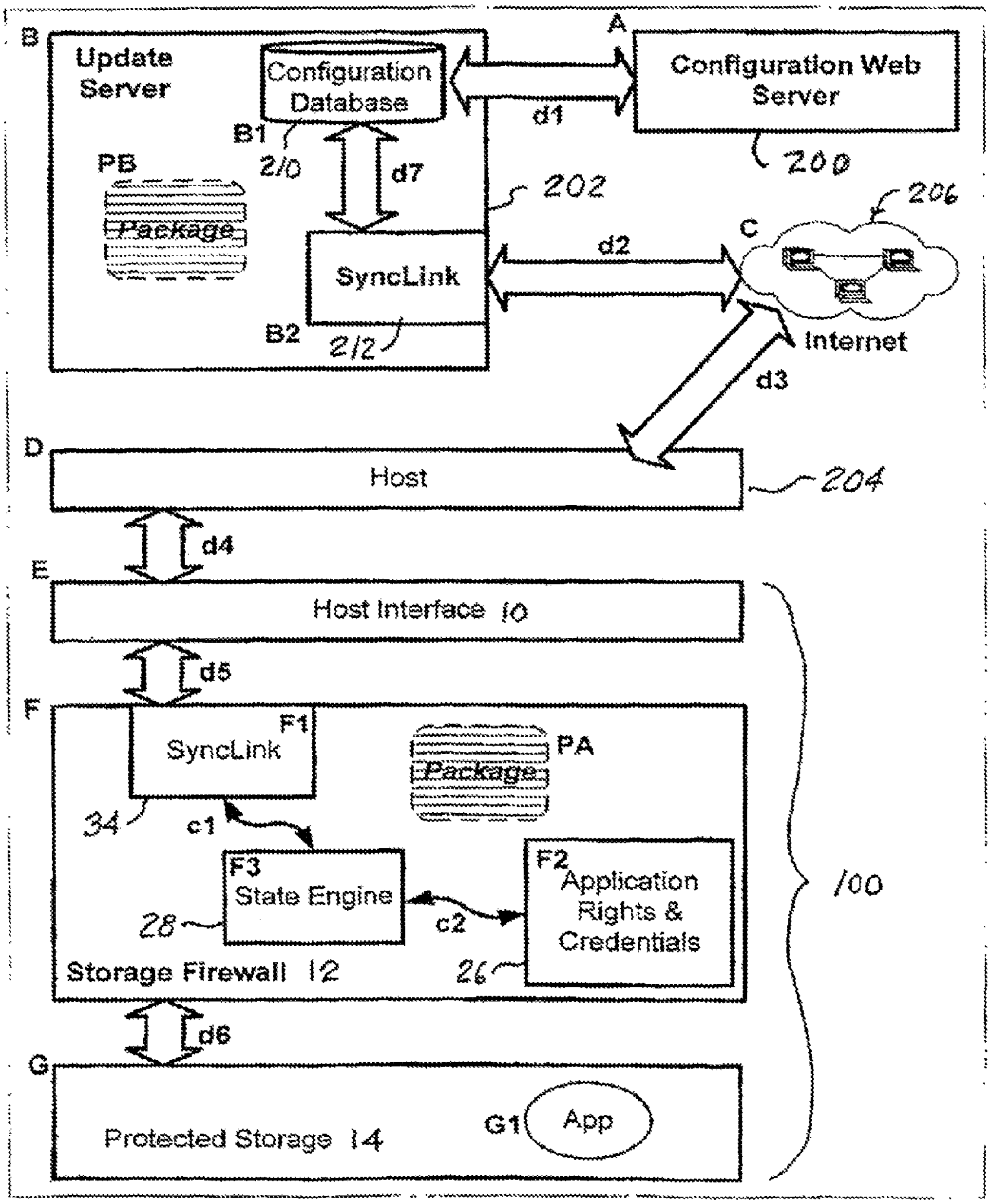
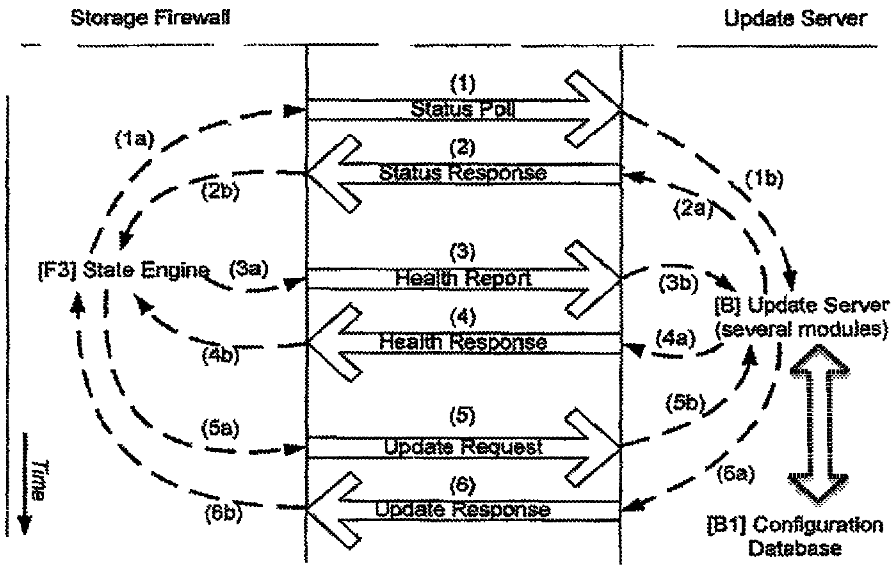


Fig. 5



Chain of Requests

Fig. 6

Solid State Disk (SSD) Conceptual Design

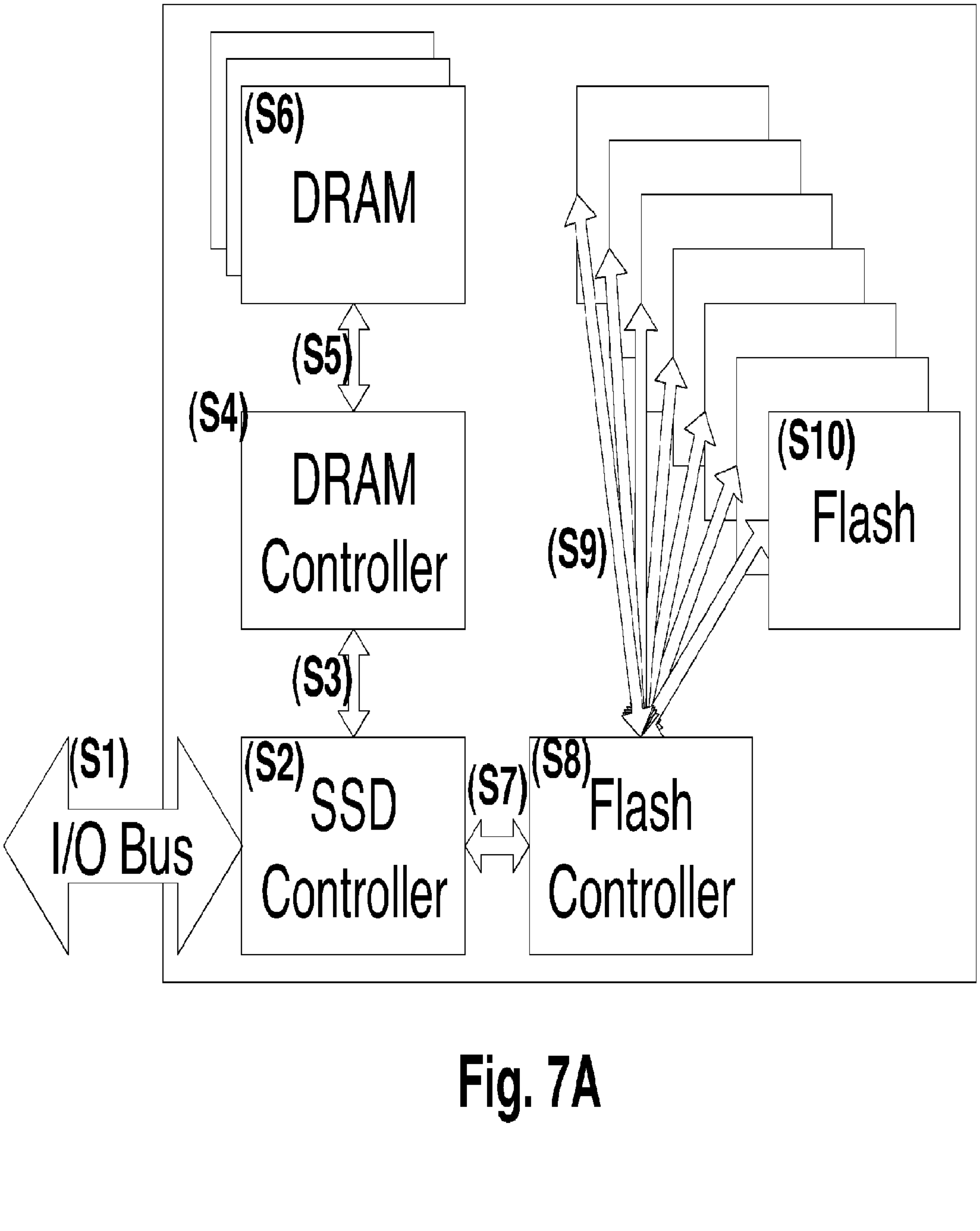


Fig. 7A

Solid State Disk (SSD) Conceptual Design with Storage Firewall Components integrated into SSD Components

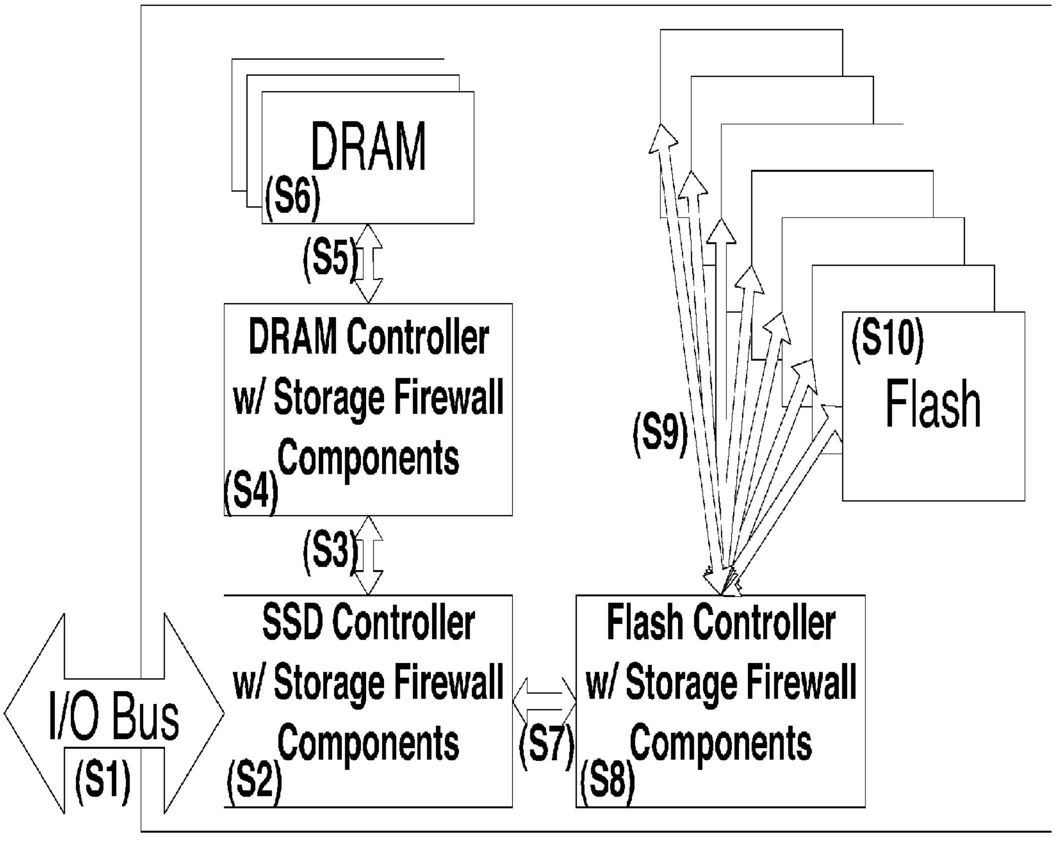
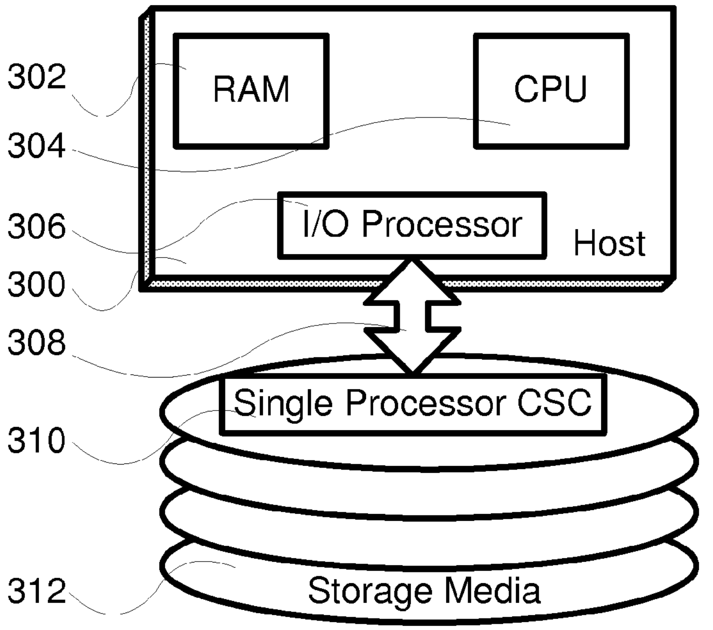

DRAM
(S6)
(S5)
DRAM Controller w/ Storage Firewall Components
(S4)
(S3)
SSD Controller w/ Storage Firewall Components
(S2)
I/O Bus
(S1)
(S7)
Flash Controller w/ Storage Firewall Components
(S8)
(S9)
(S10)
Flash

Fig. 7B



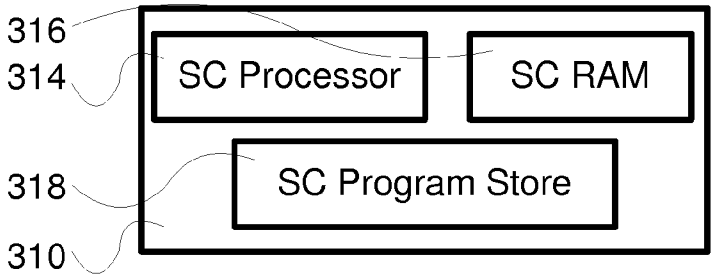


Figure 8A
Customizable Storage Controller, Single Processor

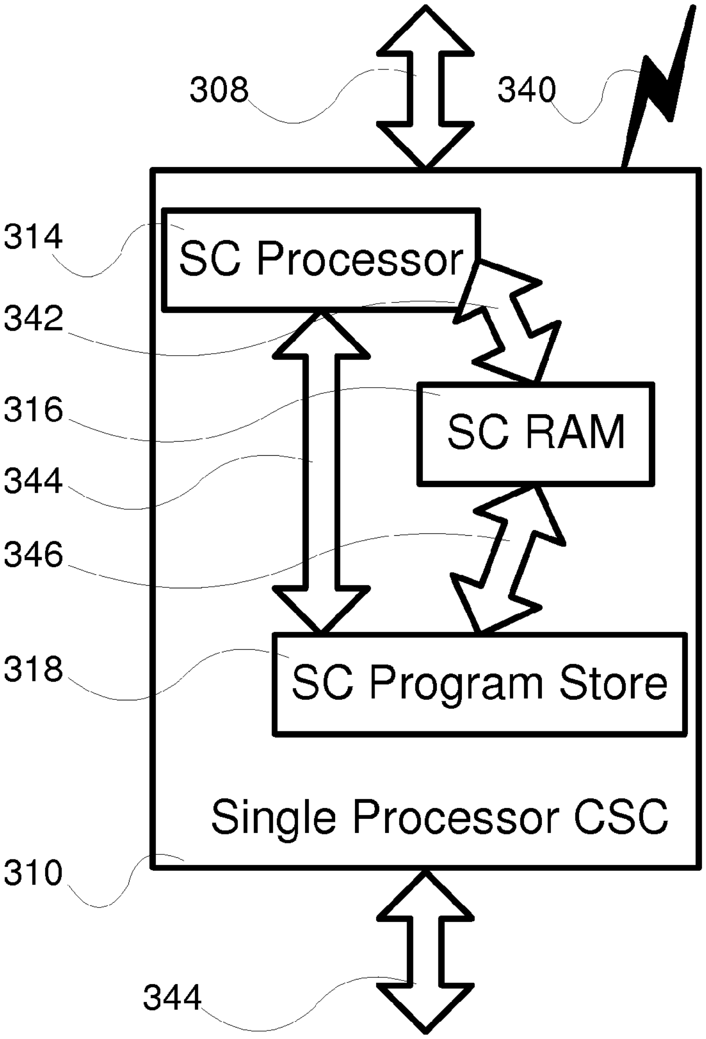


Figure 8B
Customizable Storage Controller, Single Processor, Internals

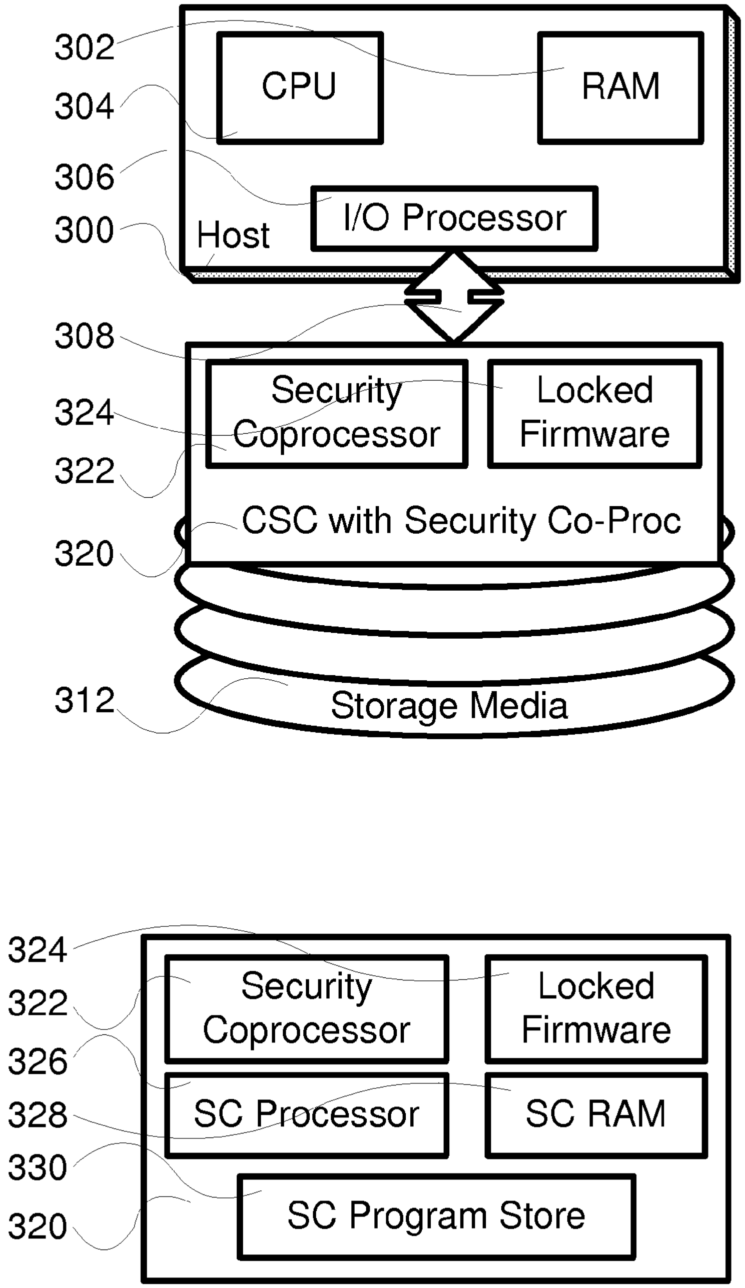


Figure 9A
Customizable Storage Controller with Security Coprocessor

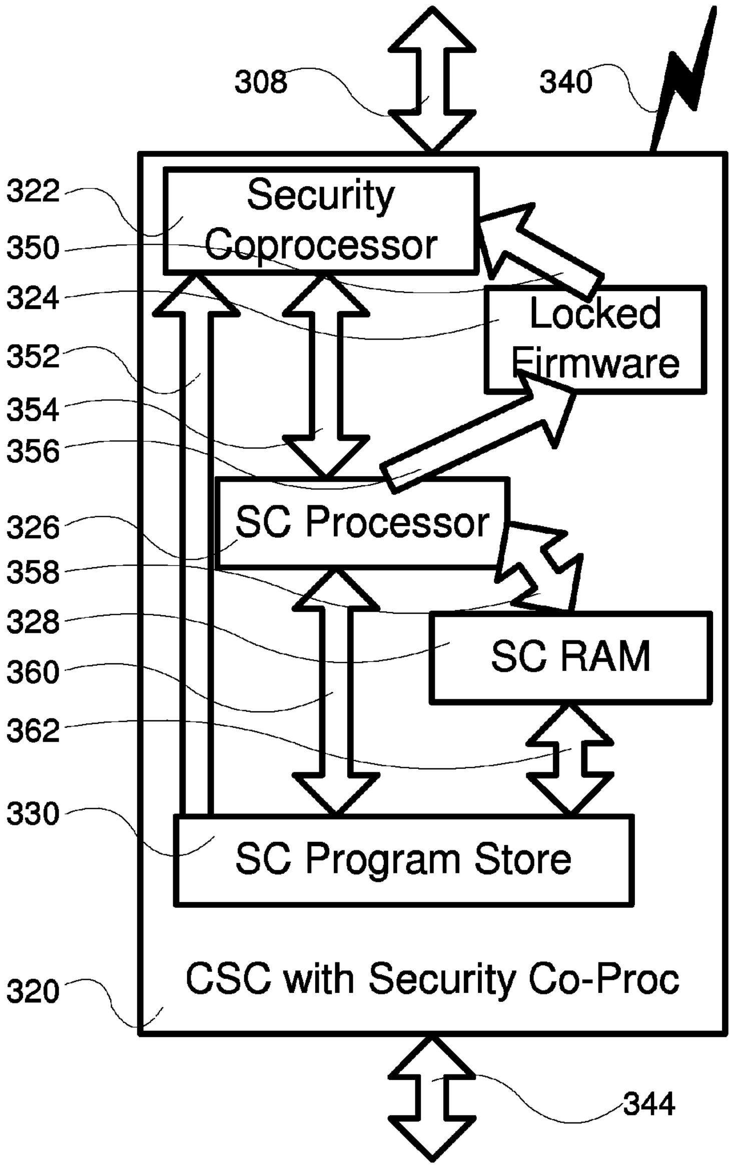


Figure 9B
Customizable Storage Controller with Security Coprocessor, Internals

CUSTOMIZABLE STORAGE CONTROLLER WITH INTEGRATED F+ STORAGE FIREWALL PROTECTION

CROSS REFERENCE TO RELATED APPLICATIONS

This application is a Continuation-in-Part of my co-pending U.S. patent application Ser. No. 12/820,137 filed on Jun. 21, 2010 and entitled FIREWALL+ STORAGE APPARATUS, METHOD AND SYSTEM, now U.S. Pat. No. 8,474,032, which is a Continuation-in-Part of my earlier filed U.S. patent application Ser. No. 11/803,947 filed on May 15, 2007 and entitled FIREWALL+ STORAGE APPARATUS, METHOD AND SYSTEM, now U.S. Pat. No. 7,743,260, and which claims priority to U.S. Provisional Application Ser. No. 60/747,536, filed May 17, 2006, entitled Storage Firewal and FlassApp Support Technology, the specifications and drawings of each such Application being expressly incorporated herein by reference.

FIELD OF THE INVENTION

The present invention relates generally to electronic data storage systems and apparatus, and more particularly to an improved storage firewall architecture, method and system that works in parallel with existing security technologies and, inter alia, provides application and system software executable authentication, user authentication & authorization in the execution of an application, examination, verification, and authentication of all storage access requests, monitoring of protected storage to detect & repair anomalous changes, encryption of protected storage, both data and software, provisioning (deployment) of patches, configuration changes, and software through a secure synchronization link to a configuration and patch management server, and server-based system administration & configuration mechanisms that support client devices by providing control of device-based system administration & configuration mechanisms, these being part of an effective system to prevent malware from penetrating local device configuration mechanisms, to cleanse local storage of malware and anomalies, and to provide other necessary maintenance services.

BACKGROUND

Network firewalls have been around for long enough that most IT professionals define them as means for protecting computers on a network by filtering at the network perimeter that which is permitted to enter a local area network (LAN) from an external network. Network firewalls are usually deployed so that they can filter all attempts to reach any of the computers on the internal network. This filtering is usually aft of the IP (Internet Protocol) packet level, though more recent technologies extend this approach to higher levels of the IP network stack. (The IP network stack is the way in which the Internet protocols are layered on top of each other in order to provide a modular design to these protocols. At the bottom is the hardware layer, and at the top is the application layer. Near the bottom are the layers that are responsible for communicating with adjacent computers on the same local network, and for tunneling through to computers and other devices on non-local networks). The trend is toward an integrated network perimeter defense system that scans and filters at all levels of the IP stack. In general concept, intrusion detection systems (IDSs) and similar defensive systems have a similar mandate and approach, but they work at higher levels of the IP network stack.

Generally, network firewalls (sometimes referred to as 'firewalls', omitting the word 'network') are deployed as hardware appliance implementations, though software implementations running on multi-purpose servers are still common. There is also a class of local software-implemented network firewalls, so-called 'personal firewalls', which sit at a computer's network interface to the external world and attempt to prevent malware and undesired network access to the protected computer. These work in a manner quite similar to the perimeter network firewalls, but with differences; such as the responsibility to monitor activities on the local computer, and to detect and filter out certain behavior such as attempts by local software to access the network.

Anti-Virus (A-V) software and local software firewalls have a somewhat similar mission, i.e., to filter out malware before it can do anything, with the assumption that most viruses these days are network borne. In addition, most desktop A-V software still check floppy disks and other removable media when they are first mounted, presumably before the possibly infected contents can do any damage. It is interesting that we still need A-V software for network access protection, since one would assume that adequate firewalls would filter out all of the attempts to penetrate the protected system, including invasion attempts by viruses.

To simplify the concept, most anti-virus software focuses on what happens to files (monitoring changes to file systems, boot sector, etc.), while network firewalls focus on network sockets (ports, addresses, packets, protocols, etc.). Of course, there are other kinds of malicious software (malware) than just viruses, e.g., trojans, backdoors, worms, etc. These can be divided into 2 sets, or rather their functionality can be considered as falling into 2 basic classes: malware that knows how to propagate itself, and malware that does not.

For purposes of this description, all of the firewall, IDS, and similar mechanisms are lumped into the general category of "firewall" since the industry seems to be going this way, with tighter integration among the various layers of perimeter defenses and internal defenses. The architecture of software that protects at the network access point(s) necessarily corresponds to the architecture of the network access protocols, so the firewalls and related mechanisms filter network traffic at the same layers (i.e. they assume the same semantics) as the protected network transport layers. This is to say that at each layer in the network stack there are layer-relevant attack vulnerabilities, and well designed network defenses have defensive elements at each layer. The defenses at each higher layer are designed, in part, to protect against the vulnerabilities of the lower layer(s). Most of the innovation in firewall design over the past 15 years has essentially been to move the analysis of layer n traffic up one level, so as to capture and analyze multiple pieces of layer n traffic, in order to assemble a more complete understanding of whether a set of layer n traffic segments corresponds to an attack using that layer.

As the defenses are gradually moved up the network stack, they must have a greater understanding of the semantics of the protected layer. To take this to its logical conclusion, since the top layer is the application, the outcome of this historical progression will be defenses that know how to protect applications; by understanding the normal state and behavior (i.e. network access, file access, disk access and on-disk presence, and memory access) in and out of the protected applications.

Malware that knows how to propagate itself falls generally into 2 behavior categories, though often malware that can do one, can also do the other. The first is inter-system propagation, such as through email systems, and the second is on-system propagation, such as copying itself into many files on a hard drive.

Roughly speaking, there are 3 kinds of anti-virus (A-V) mechanisms: network filters, file system scanners, and monitors. The network filter has an architecture roughly similar to that of the firewall architecture covered above, since it has the same goal of filtering out penetration attempts; it provides a defense against inter-system propagation The file system scanner looks for on-system viruses in all file accesses (and can be used to sieve through file sets to look for viruses, perhaps comparing against a 'signature' database, i.e., an exhaustive set of defined attributes of known virus and similar malware), while the monitor attempts to detect and block on-system viruses from doing virus-like behavior such as loading itself into memory, infecting files and operating system disk blocks, etc.

The monitor portion of an A-V tool set has an interesting challenge—ideally it would attempt to determine whether any running software is doing any virus-like behavior, but new viruses are generally able to outwit most A-V software This vulnerability to successful infections from new viruses is part of the "Zero Day" risk, There have been well documented successful Zero Day attacks from all sorts of malware, including viruses; details are available on the Internet.

With respect to the current situation of PC security defenses, although the architectures described above are interesting from a system defense perspective, it is interesting to note that even with the latest system defenses, successful malware attacks and infections are on the rise, and that installing defensive software on an infected system will not eliminate the current infection.

New, and therefore unprotected, PCs are infected within a few minutes of being put on the net, and they have to be put on the net in order to download and install the latest security defense updates and operating system security-related patches. Moreover, any PC that has been offline for several weeks is similarly vulnerable, during the time period from when it is connected to the net until the latest operating system (O/S) patches, virus signatures, and firewall updates are installed and active.

In addition, even machines that are Constantly connected, with automatic download & updates, are at risk from so-called Zero Day attacks This is because it takes time for the security patches and updates to be created in response to new malware and newly discovered vulnerabilities, and all PCs on the net during this intervening period are vulnerable to these new malware attacks. And even later, when there are available' patches and updates, with automatically scheduled downloads & updates it often takes a few days for all at risk PCs to get the latest versions.

And to make this situation even more serious, there are new threats daily. Not almost daily, as it was only a few years or so ago, but with at least one new serious threat each day. Many experts expect this situation to become worse, with new malware attack threats coming even more frequently, perhaps hourly.

As a result, one can not depend on the current generation of network and PC defensive mechanisms to prevent the infection of PCs. This situation puts at risk our data and other assets, and therefore our willingness to use the Internet.

In order to provide a secure computing environment, it is necessary to defeat malware propagation. This alone will not provide a secure computing environment, but it is a necessary element. Older security technologies are still necessary, but they are not sufficient to prevent system penetration (unauthorized access), which is why there are so many successful penetrations.

This is a significant problem. Successful penetration may leave no tracks or evidence, or a rootkit (i.e. trojan or backdoor program) may use active camouflage to prevent detection. And successful penetration leads to the installation of the virus, rootkit, or spyware, as well as to successful theft and/or damage to data.

Each day there are new attacks, new exploits, and newly discovered vulnerabilities. It is thus probably impossible to prevent zero day (brand new) attacks from successful penetration of any network connected computer systems. Given the expectation that at any given moment a computer may be infected with some variety of malware, in order to protect the confidentiality and integrity of data, it is clear that something new is needed.

The philosophy of our security model is based on the realization that locking a device down to an authorized configuration is essential for operational and information assurance—assurance that the device is performing it's expected role, not leaking information, and not permitting an attacker to inject bad information. Locking down a system in this way is very difficult or impossible using the tools that existed before the invention of the F+ storage firewall.

The security model we promote is to establish a level of assurance that the device is performing it's mission as specified, with high degrees of integrity, availability, and confidentiality. Ours may not be the only security technologies on that device, but they will establish and uphold (attempt to assure or maintain) the baseline authorized configurations that enable the device mission.

Other firms will advance all sorts of schemes to monitor the mission processor for malware and other unauthorized software, and unauthorized behavior, where their schemes inject various technologies into the software environment, likely causing just the sort of chaotic behavior they are intended to prevent, as well as device performance issues, etc. These include software reference monitors hacked into system software kernels, application level anti-virus scanners, etc. We will take the high road—focusing on better ways to protect devices and the system they are part of, with solutions that can be verified and validated in isolation before we bolt them together.

Accomplishing this goal requires, among other things, whitelist access control and filtering mechanisms (storage firewalls) on each device's storage interface(s), a flexible monitoring scheme and supporting technology on the device's interface firewalls, a configuration database and supporting technology on the server (or in the "cloud"), and a flexible and intelligent communication path between them. If it is not possible to establish a server communication link from the device's storage firewall, then this firewall has to be able to operate successfully in a standalone mode with appropriate settings and default behaviors to deal with unexpected conditions. If an anomaly (unexpected device behavior) is detected by the device itself, or one of its firewalls, or analytics on the server, then appropriate action is taken, which may include refreshing the device's internal software. This means that the storage firewall can run analysis on disk access attempts and, when judged necessary, refresh the device's mission processor by rebooting from the protected disk version of the system and mission software.

SUMMARY

It is therefore an objective of the present invention to provide an improved storage firewall architecture, method and system that works in parallel with existing security technologies and provides application software authentication, including, application registration, runtime authentication of application identity, and access control whitelisting that controls whether an executable has permission to read and write to protected storage and permission to execute (whitelisting when this software executable is installed onto, i.e. being read from, protected storage).

Another objective of the present invention is to provide an improved storage firewall architecture, method and system of the type described which provides user authentication & authorization in the execution of an application.

Still another objective of the present invention is to provide an improved storage firewall architecture of the type described which provides examination, verification, and authentication of all storage access requests.

Yet another objective of the present invention is to provide an improved storage firewall architecture, method and system of the type described which provides monitoring of protected storage to detect & repair anomalous changes. This monitoring may use any of the following techniques or algorithms: CRC (Cyclic Redundancy Check), one or more of several ECC (Error Correction Code) schemes such as the Reed-Solomon (RS) algorithm, tests of the storage firewall's file table, comparisons of Bubble contents with off-disk copies of a statistical sample of Bubble contents, when possible, search of the Bubble for malware and other unauthorized contents, and/or other algorithms and/or other forms of examinations. A further objective of the present invention is to provide an improved storage firewall architecture, method and system of the type described which provides encryption of protected storage, both data and software.

A still further objective of the present invention is to provide an improved storage firewall architecture, method and system of the type described which provides provisioning (deployment) of patches, configuration changes, and software (including application software, system software, and storage firewall security and system maintenance and management software) through a secure synchronization link to a configuration and patch management server.

An additional objective of the present invention is to provide an improved storage firewall architecture, method and system of the type described which provides server-based system administration & configuration to prevent malware from penetrating local configuration mechanisms.

Briefly, the present invention is directed to an improved storage firewall architecture, method and system that works in parallel with existing security technologies and, inter alia, provides application software authentication, user authentication & authorization and software executable authentication and authorization in the execution of an application, examination, verification, and authentication of all storage access requests, monitoring of protected storage to detect & repair anomalous changes, encryption of protected storage, both data and software, provisioning (deployment) of patches, configuration changes, and software through a secure synchronization link to a configuration and patch management server, and server-based system administration & configuration to prevent malware and other attacks from penetrating local configuration and security mechanisms, and when an attack does successfully penetrate these defenses, to detect the effects of the penetration so as to remediate and/or resolve the situation. There is a further benefit that some disk errors and some other system errors are remediated as well.

A key advantage of the present invention is that it includes a mechanism which provides application software registration & runtime authentication. Only registered (authenticated) applications can access the protected storage area.

Another advantage of the present invention is that the control and support server (update server) provides off-machine services that enable a local storage unit to verify that it has not been compromised.

Still another advantage of the present invention is that interactions between an update server and a local storage firewall can be configured as necessary for software support and maintenance, permitting the local system and/or local protected storage configuration and/or contents to be "locked down" from local changes, which prevents or ameliorates or repairs the effect of local attacks. When this update server configuration facility is not mandated by security policy and system configuration, other storage firewall functionality is still functional and can still be used.

The storage firewall in accordance with the present invention provides even stronger protection if implemented in hardware, perhaps as part of ASIC firmware because its instructions and data structures are not resident on the host (protected) computer, and are thus not accessible to malware on that computer.

A storage firewall ASIC in accordance with the present invention can be integrated directly into the control circuit of any type of storage device including a hard disk drive (HDD), be it of the rotating platters type or of solid state type (SSD, RAM, Flash, etc.) memory. As background, Solid State Disks (SSDs) have a wide variety of designs. Many are being designed using a combination of RAM, often DRAM, and Flash, where the DRAM component provides fast read/write access, and the Flash component provides non-volatility (or non-transience). A storage firewall's components will likely be integrated into both the DRAM controller and Flash controller portions of the SSD, as well as the overall SSD disk controller. Similarly, with any other current and future storage technology, the storage firewall implementation may be designed into the various parts of the storage.

Moreover, a storage firewall in accordance with the present invention can even be integrated into other memory and storage-related devices such as tape drives, RAM, etc., and will have value in any system architecture to provide protection of application software located adjacent to data used by that application.

Since the file system used within a storage unit in accordance with the present invention is different from the host PC's file system, the architecture of the storage unit can be considered an example of file system virtualization even though it is not part of a SAN storage resource pool, etc. This protected and managed "virtualized" storage can then be integrated with remote server based services such as backup, collaboration tools, file synchronization, etc., so that, in some implementations, the data is protected no matter where it is physically located, with seamless access.

An additional advantage of the present invention is that the storage firewall can be integrated with biometric or RSA-style secure ID devices such as the thumb scan type

Flash Drive devices that require a correct thumb to be identified in order to authenticate a user as authorized to use a device.

A storage firewall in accordance with the present invention can be used to filter or control access to any computer or digital storage, both long term storage such as disks, tapes, CD's, etc., and short term or temporary storage such as RAM (computer main memory). Furthermore, any digital (or analog) storage, on any digital (or analog) device, can be protected by a storage firewall, including PCs, cell phones, embedded systems, RFID chips, vending machines, space craft, war ships, servers, telephone fabric switching systems, etc.

Yet another advantage of the present invention is that a storage firewall in accordance therewith can be implemented in software, firmware or hardware.

These and other objects and advantages of the present invention will no doubt become apparent to those skilled in the art after a reading of the following detailed disclosure which makes reference to the several figures of the drawing.

IN THE DRAWING

FIG. **1** is a block diagram illustrating an embodiment of a data storage device including a storage firewall in accordance with the present invention;

FIG. **2** is a diagram schematically illustrating an exemplary embodiment of a storage device implemented in a USB flash drive in accordance with the present invention;

FIG. **3** is a diagram illustrating Authentication Data Structures in accordance with the present invention;

FIG. **4** is a diagram illustrating interaction between the Firewall=Storage Firewall and an Application stored in Protected Storage in accordance with the present invention;

FIG. **5** is a functional block diagram illustrating a Firewall+ Storage Firewall and Remote Management System in accordance with the present invention; and

FIG. **6** is a diagram illustrating communications between a Storage Firewall and an Update Server in accordance with the present invention.

FIG. **7**A is an illustration of a Solid State Disk (SSD) Conceptual Design, showing in concept how an SSD's components may be organized. (Prior Art).

FIG. **7**B is an illustration of a Solid State Disk (SSD) Conceptual Design with Storage Firewall Components integrated into SSD Components.

FIG. **8**A is an illustration of a customizable storage controller (CSC) having a single processor in accordance with a first embodiment of the present invention.

FIG. **8**B is a generalized block diagram showing the principal components of the single processor CSC of FIG. **8**A.

FIG. **9**A is an illustration of a customizable storage controller (CSC) having a security coprocessor in accordance with a second embodiment of the present invention.

FIG. **9**B is a generalized block diagram showing the principal internal components of the second embodiment of FIG. **9**A.

DETAILED DESCRIPTION

Referring now to FIG. **1** of the drawing, a data storage system in accordance with the present invention, and sometimes referred to herein as the Firewall+ storage firewall or the F+ storage firewall, is schematically illustrated in block diagram form at **100**. As depicted, the system includes a host interface **10**, a storage firewall **12** and a protected storage component **14**.

The illustrated host interface **10** provides an interface between a host computer (not shown) and a storage firewall **12**. The storage firewall **12** is composed of a transaction processor component **20**, a working memory component **22**, an encryption/decryption component **24**, and an application rights & credentials component **26**. The protected storage **14** is the memory that is being protected. The storage firewall **12** is connected to the host (not shown) through the interface **10** and sits between the interface **10** and the protected storage **14**.

Transaction processor component **20** processes storage access requests and other requests related to the administration of the storage firewall **12**, and includes a state engine module **28**, an authentication/authorization module **30**, an access controller module **32** and a SyncLink module **34**.

Working memory component **22** provides local memory storage that persists across transactions. It is used for a variety of processing memory and storage purposes by the storage firewall.

The encryption/decryption component **24** provides encryption and decryption functionality for both storage firewall processing and encryption and decryption of data of authorized transactions. It also keeps track of the symmetric key needed for the encryption and decryption operations, and provides the command & control path to the protected storage **14** from the transaction processor component **20**.

The application rights & credentials component **26** stores, processes, and provides user and application credentials and access parameters for authentication, authorization, and access control purposes.

State engine module **28** provides for execution of the transaction decision tree or state table of the storage firewall **20**. All transactions are processed through the state engine module, which decides what to do, how to respond, and which other modules and components to use to meet the requirements of the requested transaction.

Authentication/authorization module **30** provides the authentication and authorization functionality of the storage firewall. This means that the authentication/authorization module examines the current credentials and identifying information of the requesting user and the requesting application, then decides whether to grant access to the protected storage. If granted, an authentication token is generated.

Access controller module **32** provides access control for an application that requests access to the protected storage **14**. The requesting application must provide a valid authentication token. The access controller module validates and verifies this authentication token. If the authentication token is valid, the request is evaluated to decide whether to grant access, based on previously stored authorization parameters.

The SyncLink module **34** provides an intelligent communication channel to a remote management system's update server (not shown). As indicated above, the SyncLink module is part of the transaction processor **20**. Although the storage firewall can operate as a stand alone storage unit and provide its security functionality without a SyncLink component, as will be discussed in detail below, a Firewall+ storage firewall without the SyncLink module **34** will not be integratable with a remote update server (not shown) in that it will not be able to obtain updates, such as changed and new application software.

The protected storage unit **14** is the memory that is being protected by the firewall **12** and may contain protected applications **40**, protected files **42**, and protected storage firewall management data **44**. Applications are stored as files **42** and management data may also be stored as files. This is to say that the files **42** may include any digital information or object.

The double headed arrow d**1** represents the data and request signal path between the host Interface **10** and the transaction processor component **20** in firewall **12**. It is also used as a path into and through the host Interface **10** (and host computer's Internet connection) to poll a remote management system update server (not shown) for updates.

The double headed arrow d**2** is the data path between the transaction processor component **20** and the encryption/decryption component of the firewall **12**. Data moving on the path d**2** has not been encrypted by the encryption/decryption component.

Double headed arrow d**3** is the data path between the encryption/decryption component **24** and the protected storage **14**. Data on the path d**3** has been encrypted by the encryption/decryption component **24**.

The arrow c**4** is the control connection and data path between the authentication/authorization module **30** and the encryption/decryption component **24** and carries commands to encrypt and un-encrypt (clear) data to the encryption/decryption component **24**. c**4** also carries encrypted data back to the authentication/authorization module **30**. In addition, c**4** also carries commands to decrypt and encrypt data to the encryption/decryption component **24**, and carries decrypted data back to the authentication/authorization module **30**.

Arrow c**5** is the control connection and data path between the access controller module **32** and the encryption/decryption component **24**. c**5** carries commands to encrypt and un-encrypt (clear) data to the encryption/decryption component **24**, and carries encrypted data back to the access controller module **32**. c**5** also carries commands to decrypt and encrypt data to the encryption/decryption component **24**, and carries decrypted data back to the access controller module **32**.

The arrow c**6** is the control connection and data path between the authentication/authorization module **30** and the application rights & credentials component **26**. c**6** carries user and application credentials and access parameters for authentication and authorization processing; and also carries authentication tokens that represent granted permissions for access requests.

c**7** is the control connection and data path between the access controller module **32** and the application rights & credentials component **26** and carries access parameters and authentication tokens for authorization and access control processing. c**7** also carries granted permissions for access requests.

Arrow c**8** is the control connection and data path between the transaction processor component **20** and the working memory component **22**. It carries a wide range of data in support of storage firewall processing.

c**9** is the control connection and data path between the application rights & credentials component **26** and the working memory component **22**. It carries data in support of authentication, authorization, access control, application and user rights and credentials processing.

Arrow c**10** is the control connection and data path between the application rights & credentials component **26** and the encryption/decryption component **24**. c**10** carries commands to encrypt and un-encrypt (clear) data to the encryption/decryption component **24**, and carries encrypted data back to the application rights & credentials component. c**10** also carries commands to decrypt and encrypt data to the encryption/decryption component **24**, and carries decrypted data back to the application rights & credentials component **26**.

c**11** is the control connection between the state engine module **28** and the encryption/decryption component **24** as well as control signals from the state engine module **28** to the protected storage **14** by passing them to the encryption/decryption component **24**, which in turn passes them through to the protected storage **14**.

Arrow c**12** is the control connection and data path between the state engine module **28** and the authentication/authorization module **30**.

Arrow c**13** is the control connection and data path between the state engine module **28** and the access controller module **32**.

And finally, arrow c**14** is the control connection and data path between the state engine module **28** and SyncLink module **34**.

The illustrated storage unit undergoes several phases of operation:

a). Quiescent Phase (Pre-Startup, Not Running)

When the device is in its Quiescent Phase, the protected storage is not reachable, i.e. it can not be written to, nor can it be read from.

b). Startup Phase

When the storage device starts running, it is in its Startup Phase. In the startup phase the device becomes ready to handle local transactions. But the storage firewall **12** will not grant access to the protected storage **14** until there are valid credentials received and stored by the application rights & credentials component **26**. It is required that these valid credentials be successfully provided before storage access transactions can be granted.

c). Active Phase

The Active Phase begins when the storage firewall **12** begins to grant access to the protected storage **14**. During the active phase, the storage firewall **12** requests updates from an update manager (not shown) if a secure connection can be established.

d). Shutdown Phase

The Shutdown Phase begins when the storage firewall **12** stops granting access to the protected storage **14**, and performs cleanup operations such as deleting obsolete credentials from the application rights & credentials component **26**. This phase is not necessary to the successful subsequent operation of the storage device. If the shutdown phase cleanup is not performed, then these actions might be performed on the next startup of the storage device.

Implementation

As pointed out above, the transaction processor component **20** has three subsystems. The implementation of the transaction processor component involves the integration of three subsystems; i.e., the state engine module **28**, the authentication/authorization module **30**, and the access controller module **32**. As will be further described below, the SyncLink module **34** is also included for enabling communication with a remote server.

The state engine module **28** is the transactional portion of transaction processor **20**, and is based on a software coded state table, an implementation of the storage firewall decision tree. To save space on the chip, the decision tree is encoded in a state table. The state table has an entry for every transaction conducted by the storage firewall unit.

There is also one or more no-op transactions provided as catch-ails for received transactions that do not match any of the supported transactions. This no-op transaction ability is an important security measure, responding to attempts to map the transaction set provided by the specific storage firewall being attacked. The no-ops in the storage firewall's transaction processor component's state engine module may or may not provide deliberately erroneous results (obfuscation) to the attacker, or may ignore the request; this ambiguity is part of the defense mechanism.

If the chip is implemented as an FPGA (Field Programmable Gate Array) or other field-changeable chip or chip portion, it may be possible to upgrade or replace the state table in an operational storage firewall's transaction processor component's state engine module. This is useful to add transactions, or to improve or change the behavior of existing transactions. It can also be a security vulnerability, if an attacker is able to change transaction behavior. This implementation is preferred in some consumer products, to reduce product support costs. In addition, upgrading the state table permits completely new functionality to be added to the storage firewall, as well as permitting existing functionality to be removed. If new states (and the corresponding operations) are added to the state table, then additional corresponding executable software may also be added, or the new states may activate pre-deployed executable software, or may apply existing executable software in a new way. Executable "software" may be hardwired into the chip used for the storage firewall implementation, or may be pre-deployed in some other way, or may be provisioned (deployed) from (by) the update and configuration server. If a state table operation can not be performed as a result of the absence of the corresponding executable software, the resulting action shall be as if the state engine directed the processor to a no-op transaction.

If the chip is an ASIC (Application-Specific Integrated Circuit) or other non-field-changeable chip or chip portion, then it is not possible to upgrade or replace the state table in the field. This implementation is preferred in a highly secure product or installation, such as a military application.

The state engine module uses the authentication/authorization module to evaluate (verify and validate) input application-signatures, user input credentials, and other authentication and authorization inputs. If an authentication token is generated as part of the initiation request transaction processing, it is provided to the application through the host interface.

The state engine module **28** uses the access controller module **32** to evaluate (verify and validate) input authentication tokens. Inputs to the state engine module are provided by the host interface **10**. These inputs are part of transaction requests. If the inputs are valid, and the internal state permits, the requests are granted.

For granted (authorized) read transactions, the state engine module directs the encryption/decryption component **24** and the protected storage component **14** to provide and decrypt the requested data. The data is provided to the application through the host interface.

For granted (authorized) write transactions, the state engine module directs the encryption/decryption component and the protected storage component to accept and encrypt the provided data. The data is provided to the storage firewall from the host interface.

User authentication and application authentication are implemented differently. As used herein, the term authentication refers to both authentication and authorization.

The authentication/authorization module **30** is controlled by the state engine module **28**. Identification credentials of the current user are used to authenticate this user. The provided user credentials are validated by the authentication/authorization module. The implementation is to use the application rights & credentials component **26** and the encryption/decryption component to compare the provided user credentials against those stored previously.

Access to the protected storage **14** is granted only if the provided user credentials are successfully validated. The result determines whether the current user will be permitted to access the protected storage. The application rights & credentials component **26** indicates whether or not the provided user credentials are validated.

The authentication/authorization module **30** uses the application rights & credentials component **26** to keep track of whether valid user credentials have been provided by the user since the storage firewall started running. A fresh user authentication must take place each time the storage firewall starts running.

The authentication/authorization module also uses the application rights & credentials to store, process, and retrieve the user's authorization rights. When an application wants to access the protected storage, it must request initiation of a storage firewall transaction session. The application provides an application-signature with the initiation request.

The authentication/authorization module **30** uses the application rights & credentials component **26** and the encryption/decryption component **24** to attempt to validate the application-signature. If the application-signature is valid, and the current user credentials are valid, then the application rights & credentials component is used to generate an authentication token. The authentication/authorization module provides the authentication token to the state engine module **28**.

The access controller module **32** is controlled by the state engine module **28**. Each access request by an application must have a valid authentication token (other than the initiation transaction). The access controller uses the authentication token provided by the application to evaluate whether to permit the current transaction request.

The access controller module **32** uses the application rights & credentials component **26** to validate the authentication token, and to retrieve the application's rights. For a valid authentication token, the application rights govern whether to approve the request.

The SyncLink module **34** is controlled by the state engine **28**. Periodically, the state engine uses the SyncLink module to poll the update server (not shown) of a remote management system (not shown) for updates. When an update is received, the SyncLink module opens up the retrieved update package, and provides the contents of the update package to the state engine for processing.

In the illustrated embodiment the working memory component **22** is implemented as solid state memory, some combination of RAM and/or Flash memory, or some other solid state memory. The encryption/decryption component **24** is implemented using the encryption algorithm AES (Advanced Encryption Standard); a US government encryption standard, which is a version of the Rijndael block cipher. It is documented by the US National Institute of Standards and Technology (NIST) as U.S. FIPS PUB 197. The encryption/decryption component provides encryption and decryption operations that can be used by other storage firewall components and modules. It also encrypts and decrypts data as it moves into and out of the protected storage component.

The application rights & credentials component **26** is implemented as a set of data structures and operations that act on these data structures. It uses the working memory component **22** to store the data structures. The data structures include an application rights table, a credentials table, and user registration data (which are further described below). The component **26** provides operations that validate provided values against stored values.

When validating user registration data, the validation method for the user id is to compare the provided user id against the stored user id, byte for byte. The validation method for the password is to create an encrypted hash of the provided password, then to compare this against the stored password, byte for byte.

When validating an application-signature, the validation method is to create an encrypted hash of the provided application-signature, then to verify that this matches the corresponding value stored in the application rights table for the application that submitted it with the current initiation request.

When validating an authentication token, the validation method is to create an encrypted hash of the provided authentication token, then to use this encrypted hash to verify that the provided authentication token is current, refers to an active entry in the credentials table, and was assigned to the application that submitted it with the current request.

In this embodiment the protected storage component **14** is implemented as NAND flash chips.

Explanation Of Operations

Major transactions are described below in relation to the functional diagram. These transactions are:

Initiation;

Open file for read; and

Request and receive update for application.

All transactions come to the transaction processor component **20** via the host Interface **10**, and all transaction results are returned (sent back) through the host Interface. The portion of the transaction processor component **20** that actually handles all transactions is the state engine module **28**. This module provides all transaction command and control, using other storage firewall elements as needed.

Transaction: Initiation

The application sends an initiation request transaction to the storage firewall through the host Interface **10** to the transaction processor component **20**, or in actuality, to the state engine module **28**. The state engine uses the authentication/authorization module **30** to validate the transaction parameter application-signature, and to provide an authentication token.

The authentication/authorization module **30** uses the application rights & credentials component **26** to validate the application-signature against values stored in the application rights table, to verify that the current user is properly logged in to the storage firewall, to add an entry into the credentials table, and to generate the authentication token.

The application rights & credentials component **26** uses the encryption/decryption component **24** to hash and encrypt the authentication token, to store a hashed & encrypted copy in the appropriate entry of the credentials table.

Transaction: File_Open_Read

In this transaction, the application sends an open file for read (File_Open_Read) request to the storage firewall which passes through the host Interface **10** to the transaction processor, or in actuality, to the state engine module **28**. The state engine uses the access controller module **32** to validate the authentication token, and to provide the application rights. The access controller module **32** uses the application rights & credentials component **26** to validate the authentication token, and to provide the application rights.

The application rights & credentials component **26** uses the encryption/decryption component **24** to hash and encrypt the input authentication token, for comparison against a copy stored in the appropriate entry of the credentials table.

If the application is authorized to open that file, then the state engine module **28** adds an entry to the File Table, and generates a file reference.

If the file has to be created, then the state engine module creates the file through the encryption/decryption component **24**. The state engine module then returns the file reference to the application through the host Interface **10**.

Transaction: Request and Receive Update for Application

As previously indicated, the state Engine **28** uses the SyncLink module **34** to poll a remote management system's update server for updates by sending an Internet update poll request packet to the remote update server. In response, the update server sends an update package to the local host computer.

The state engine module **28** uses the encryption/decryption component **24** to validate the retrieved update package by cryptographic means. The state engine uses the SyncLink module to open the retrieved update package, and provide the contents of the update package to the state engine module for processing.

The package may, for example, contain a software application and associated application rights and credentials data; if so, then the state engine uses the encryption/decryption component to encrypt the application and install it into the protected storage **14**, and the state engine uses the authentication/authorization module, which uses the application rights & credentials component **26**, to install the associated application rights and credentials data into the corresponding data structures.

FIG. **2** is an artistic or conceptual diagram schematically illustrating at **100** an exemplary embodiment of a storage device implemented in a USB flash drive in accordance with the present invention. In this diagram a host interface is shown at **110**. The host interface **110** is a USB controller. USB controller chips and designs are commercially available from manufacturers such as Cypress, Anchorchips, Scanlogic, and Intel.

The active Firewall+ Storage Firewall is shown at **112***a*, and the storage component is shown as a Bubble at **114**. If the implementation of the Firewall+ is in "software" such that the Firewall+ Storage Firewall executable is stored within the Bubble (as shown at **112***b*), the Firewall+ execution may be as a process executed on the host computer, rather than in the USB flash drive.

In most cases, Bubble contents will be organized in a Bubble-internal file system. This includes error detecting data, error correcting data, application rights table, Bubble maintenance data, and self-referential hash values. All of these are referred to as error correcting data. Since even the Bubble's own maintenance data is stored in a Bubble-internal file, the Bubble access operations are easier to implement, maintain, and test.

On the other hand, if the Firewall+ Storage Firewall is implemented on a chip (a "hardware" implementation) in the USB flash drive, then the "Stored Firewall+" **112***b* may not be present inside the Bubble.

Other possible implementations may put the Firewall+ on other hardware, such as the host computer's main board or storage controller. In these implementations, the Firewall+ may be conceptually portrayed as part of the host computer rather than a part of the USB flash drive.

Also, the same design may be used for storage devices and media other than USB flash drives.

As will be understood from the above more generalized description, the storage firewall unit has several inter-operating parts. The focus of this discussion is on aspects of the design that permit construction (implementation) of various embodiments. As previously suggested, there will be multiple different implementations and embodiments of this design to suit various applications Moreover, there will be permutations required by the architectures of the products and technology (hardware and systems) being augmented by the Firewall+ functionality, and by the market requirements of the resulting improved products. However, these embodiments will all share the design philosophies, elements, and the architecture described herein.

This description will include various components and elements of the embodiments, both inside the Bubble and in the Firewall+ storage firewall, then describe the Firewall+ storage firewall in action. It is to be understood however that all transactions will not be covered since the focus is on those transactions that demonstrate the design of the present invention.

The authentication and authorization architecture will also be discussed since authentication, authorization, and access control (AAA) are key elements to any security-enabled system; Firewall+ is no exception. There will no doubt be some repetition of material covered earlier, but in general, this section goes into greater depth, focusing on the default algorithms. It will be appreciated that the AAA module can be replaced by a third party solution (and may well be for some implementations), but the default methods should be adequate for most purposes.

An overview of the principle transactions will also be discussed as implemented in a "state table"—the design does not require that every implementation use a state table for its transaction decision tree, but the documented state table is useful as a summary of important transactions. As in the above, not all transactions are covered in this table.

In most cases, Bubble contents will be organized in a Bubble-internal file system. This includes error detecting data, error correcting data, application rights table, Bubble maintenance data, and self-referential hash values. All of these are referred to as error correcting data. Since even the Bubble's own maintenance data is stored in a Bubble-internal file, the Bubble access operations are easier to implement, maintain, and test.

The following terms will be used in this discussion. If a term is used that is missing from this "glossary", it is expected to have the meaning assigned elsewhere herein, or if not specifically defined herein, it should be considered to have the meaning provided by a standard published authoritative technical dictionary.

Glossary	
Access control authorization	The authentication token is provided back to the Firewall+ Storage Firewall with every subsequent access request. The authentication token is cryptographically validated on every use, and then used to validate access permissions for the requesting application.
Actor	a term which has become common in the IT industry for an entity that acts upon other entities. An actor can be a person, a software executable, a hardware device, et alia. From Wikipedia, the free encyclopedia, June 2010: "In computer science, the Actor model is a mathematical model of concurrent computation that treats "actors" as the universal primitives of concurrent digital computation: in response to a message that it receives, an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message received. The Actor model originated in 1973."
Authentication	Part of the Initiation transaction: when an application starts to run, it must first prove it is the corresponding application, among those in the Rights table, thereby proving that it has rights to access the Bubble. This proof is provided by acryptographic operation involving the application supplied application-signature.
Authorization	Part of the Initiation transaction: when an application starts to run, the rights are examined, a credentials entry is made, and an authentication token is generated. The authentication token links to the entry in the credentials table, which provides a link back to the rights table.
Application	Application software that intends to access a "Bubble". This might be non-application software such as host system software. Usually used as a synonym for software executable. An application can also be thought of as an instance of executable software.
Blacklist	a term which has become common in the IT industry for a list of software executables or some other actors that are specifically not authorized to do something. Often used to refer to a list of software executables believed to be malicious software, malware. The opposite of a whitelist. The prior art includes many anti-virus software tools that rely on a blacklist of virus signatures.
Bubble	Protected storage. This might be a single partition, a set of volumes, or any other defined storage area.
Event	A message that links to a trigger, elicited by an internal or external situation or condition. State changes are usually caused by events. The event can 'trigger' a reaction response by the storage firewall. (more in Trigger)
Firewall+	The Firewall+ Storage Firewall executable and process. In some cases this is referred to as the F+ int or Firewall+ int (int for internal) because in some implementations the executable will be stored within the Bubble when not active. But other implementations of the Firewall+ will be on an ASIC, or FPGA, not stored within the Bubble. The Firewall+ term may also sometimes be used to refer to the entire Firewall+ architecture.
Firewall+ Startup-Diagnostics (FSD)	In some implementations, the FSD executable will be executed before the Firewall+ storage firewall executable is activated; it verifies the health of the Bubble and Firewall+executable, repairing damage, or attempting to repair damage, if found. In some cases this is referred to as the F+ ext or Firewall+ ext (ext for external) because its executable is never found within the Bubble. The FSD executable can also be used to perform its health and malware scan at other times than when starting up the storage firewall. In some implementations the FSD components will be designed into HDD controller components or SSD controller components.
Initiation	Handshake transaction when an application starts running, informing the Firewall+ that the application will start sending access requests.
Registration	Firewall+ operation that adds applications to the Rights table; part of provisioning
Rights table	Persistent data structure that contains application rights and credentials for registered applications. In some cases this is referred to as the registration table or registry.
Trigger	A term used to indicate that a state transition is being caused by a stimuli, often an stimuli external to the storage firewall. Events link to triggers. A situation or condition has elicited from a component of the storage firewall the generation of an event; the event message is sent from one storage firewall component to another, causing a state change, and possibly causing the storage firewall to act on an external entity, where this action can be the generation of a 'warning' message to the support server, and/or the activation of a locally present executable.

-continued

Glossary	
Whitelist	A term which has become common in the IT industry for a list of software executables or some other actors that are authorized to do something. The storage firewall list of authorized executables can be thought of a type of whitelist. The opposite of a blacklist. The whitelist concept can be thought of in this way: if the of names of the people invited to a party are on a list, then that list is a whitelist.

Events and Triggers—new terms—conditions can cause an 'event', the event can 'trigger' a reaction response by the storage firewall; this response can be the generation of a 'warning' message to the server, and/or the activation of a locally present executable.

Application Rights Table (aka Rights Table, Registration Table)

An application rights table is illustrated in FIG. **3** and is used to register deployed applications when provisioned (deployed or installed) to the Firewall+ storage firewall protected storage or device. If there are no applications within the Bubble, the application rights table does not have to exist—it is created when the first application is provisioned. It does not have to be destroyed when (or if) all applications are deleted from the Bubble.

In general, the application rights table keeps track of the presence of provisioned application software and the rights software applications have to access the Bubble. Software does not have to be installed within a Bubble to have rights to access that Bubble, but generally installation within a Bubble confers associated access rights. When an application is installed within a Bubble, or a software executable has been deployed as part of the storage firewall itself, the information in the rights table includes execution-related controls an attributes such as scheduling, execute permissions, required user or role authorizations, etc., as well as storage access permissions.

The application rights table is a key data structure. It will grow as applications are added to the Bubble. It can be indexed by the application identity (a hash of the application's version and serial_number) as well as by a generated authentication token (described below).

Key elements of the application rights table include data fields for:

- version—version of the currently provisioned application, used for several purposes such as optional replacement if application is damaged or lost, as well as update and patch. May be linked to a general update behavior, on the update server.
- serial_number—string identifying the currently provisioned application, generally keyed to a specific license stored on the update sever. It is used for several purposes such as replacement if application is damaged or lost, as well as update and patch.
- authentication—flag if this application is permitted Bubble access, plus other data used in the application authentication operation
- authorization—optional Bubble access permissions, other optional authorization data

The concatenation of version and serial_number is guaranteed to be a unique value across all possible supported applications. A hash of the version and serial_number may be used as an index to the Rights table.

Directory Structure

The Bubble has its own file system—independent of the file system of the host computer. A POSIX compliant approach may be used in this file system. It has a hierarchical structure, not very different from the Unix directory structure, using files to store directory information, and representing files by file names, linked to file inodes. The inodes can then be used as unique file (and directory) identifiers in the File Table.

Minimal Bubble

The minimal contents of the Bubble is a Bubble empty of any application data, but with error correcting data in one or more files. The minimal Bubble will contain an empty Rights table.

The error correcting data can be used to verify the health of the Firewall+ storage firewall executable. It can also be used to validate the empty state, when the Firewall+ storage firewall responds to a query that would otherwise return Bubble contents or Bubble state.

Bubble with Data Files Only

The Bubble either has contents or does not have contents. If it has contents, the simplest case is externally referenced data plus the contents of the minimal Bubble. The externally referenced data is organized as one or more files in the Bubble-internal file system. There is error-correcting data kept, in separate storage, for each data file.

To create a Bubble-internal file, the external software (typically a software application) sends a file creation access request to the Firewall+ storage firewall. This request provides an externally referenced file name. This name maps to a Bubble-internal file reference, such that later access queries will map to the correct file. The Bubble-internal file reference is very similar to a Unix-style inode.

With One or More Applications and Associated Data

When applications are kept within the Bubble, they are stored and treated like data files, in the manner described above.

The method employed by the user to start executing a Bubble-resident application depends on the implementation of the Firewall+ storage firewall, on its context of use.

In a 'preferred embodiment', after the Firewall+ storage firewall finishes its startup operation, it will kick off a default application, an application-browser. This will permit the user to select an application from a list or menu.

In this embodiment, the application-browser will always be present in every Bubble, so they would never be completely empty. This has implications for the Rights table, etc.

Firewall+ Storage Firewall Data Structures

File Table (for Open Files)

The File Table is not a permanent structure—it is transient, created and expanded as needed.

The Firewall+ uses the File table to keep track of which files applications have open. There may be all of the expected file operations; e.g. files can be created, opened for reading or writing, closed, deleted, etc. The File Table can be cached within the Bubble or kept in transient memory (RAM), or some combination of these.

The File Table is indexed by a file reference value, or a value derived from the file reference value. (In some cases, the file reference value may be referred to as a file pointer or fp.) The file reference is passed to the application when a file is successfully opened, and is passed from the application to the Firewall+ storage firewall with every successive file operation on that open file, until it is closed.

An application may have more than one file open. For each open file there is a different file reference value.

In some other file systems it is possible for more than one application to have write access to the same file, but in this design only one application may have write access to a file at a time. Multiple applications may have simultaneous read access to a file. The way the File Table keeps track is discussed below.

The File Table keeps track of which application(s) currently have authorization (permission) to access a file, a subset of the applications that currently have permission to access the Bubble.

The File Table also keeps track of which applications have opened a file for access. Because multiple applications may have read access to a file, plus another application may have write access, it is necessary for the File Table to have an expandable structure to store references to these applications.

The authentication token is described below, but a brief mention of how the authentication token relates to the File Table is useful. Each application that is granted access to the Bubble has a unique auth_token. This authentication token is used for several purposes related to access control, but it can also be used to refer back to the corresponding application. The File Table's mechanism for tracking which applications have active file references to a file uses the applications' authentication tokens (auth_tokens) as reverse references to the applications. In case there is a need to look up application access privileges or other attributes, the tracked authentication token can be used.

There is a limit to the number of files that can be open at any one time: If the Bubble is full, and Firewall+ storage firewall's working memory is full, then additional files can not be opened, because there is no space to expand the File Table. One workaround is to reserve a portion of the Bubble for File Table expansion, but this does not eliminate the issue—there is still a limit to the number of files that can be opened at any one time. This limit depends on several variable factors, so it is not a hard number. Of course, if the Bubble were full, there would be no place to put a newly created file, so File Table expansion would be irrelevant in that case.

Authentication Token

The authentication token is provided to applications when they open contact with the Firewall+ storage firewall (the Initiation transaction)—IFF they are authenticated and authorized for Bubble access. The authentication token is used to index the Credentials table to verify application rights (privileges) for an operation such as opening a file. As such it is passed to the Firewall+ storage firewall with every application file access transaction.

For the detail minded, the authentication token, auth_token, is a hash of validating data and an index to the appropriate entry in the Credentials table.

Credentials Table

The Credentials table also shown in FIG. **3** is transient, created as needed by the Firewall+ storage firewall, generally as part of the reaction to an initiation transaction request.

The Credentials table stores a version of application authentication, authorization, and access credentials and rights. It is indexed by the authentication token. It contains an index into the Rights table for the corresponding application.

The Firewall+ Storage Firewall in Action

This portion of the disclosure is organized by transaction types, where for each there is roughly the same organization of information. The transaction types covered are:

Deployment (provisioning) and registration of the application into the Rights table.

Firewall+ startup, a sort of self-transaction, which might also kick start an application.

Firewall+ initiation (handshake) transaction when the application starts running.

Application access, with a focus on read/write requests.

For each of these, there are generally four parts:

Request/response protocol, even though the application shouldn't see most of these, as they'll be hidden within the ABI

Possible code changes to the application

Changes to the Rights table

Other components used and/or changed, if any

Overview of Application to Storage Firewall Interaction

This is an overview of the interaction between applications and the Firewall+ storage firewall, using as an example a sample set of transactions that include opening a Bubble-protected file for reading, and making a read request from that file. It provides context for the following sections.

With reference to FIG. **4** note that: the Firewall+ API initiation request is in the application's startup code; the application is linked to the Firewall+ ABI, and the application is registered in the Firewall+ Rights table.

Steps in the file access interaction. Each pair of steps (such as steps 3 & 4) is a complete request/response transaction.

1. App starts to run, and initiates contact with the Firewall+ storage firewall.
2. Firewall+ authenticates the App, and provides authorization credential (auth_token), which the Firewall+ ABI keeps track of.
3. The App sends 'open for read' file access request to F+, where the authentication token is added to the file open request by the F+ ABI
4. Firewall+ optionally logs the access request, then verifies the App's auth_token, opens the file for read, then returns a file reference—an index into the File Table
5. The App sends a read access request to Firewall+, where the authentication token is added to the file read request by the Firewall+ ABI, and the file pointer (fp) is a Firewall+ file reference, an index into the File Table.
6. Firewall+ optionally logs the access request, then verifies the App's auth_token, reads requested data from file, updates entry in the File Table indicating current position in the file, then returns the requested data to the App.

The Firewall+ ABI

The Firewall+ ABI hides the transaction details from the application. Application software that has been modified to work with the Firewall+ storage firewall will have been linked against the Firewall+ ABI, and have a small addition to its startup routine.

The Firewall+ ABI replaces the file operation library that the application was previously linked to. The Firewall+ ABI provides a file access operations set that has the same semantics and syntax, but that knows how to negotiate access with the Firewall+ storage firewall. There is an include (header) file set that matches the Firewall+ ABI.

The purpose of the Firewall+ ABI software is to provide file access operations, plus to negotiate with the Firewall+ storage firewall. This includes initiating interaction with one or more currently running Firewall+ storage firewalls, and authenticating the right of this application to access the protected storage; this is provided by the Firewall+ initiation transaction. The Firewall+ initiation transaction is performed by a function call, added to the application's startup code.

It is possible that more than one Firewall+ storage firewall may be running at the same time, and it is possible that an application may be separately authenticated to access multiple of the protected storage areas.

There will be a unique authentication token (referred to as auth_token) provided for each application-to-Firewall+ execution instance, passed to the application by a successful Firewall+ initiation transaction. The authentication token must be passed back to the Firewall+ storage firewall for every subsequent file access transaction. The Firewall+ ABI adds the authentication token to the file access requests on behalf of the application.

Application Provisioning (Deployment, Registration into the Rights Table)

A relevant transaction occurs when an application is provisioned to the protected storage.

Before dealing with this transaction, the definitions of several terms will be resolved: provisioning, deployment, and registration. In particular, there may be some confusion over the difference between provisioning and deployment. In the Firewall+ architecture, these terms may include the following meanings:

Provisioning provides the functionality, whereas

Deployment delivers and installs the mechanism (such as the application software)

In the Firewall+ design, the successful installation of application software to a Firewall+ Bubble has these steps:

Deployment of the application software

Registration of the application into the Rights table

From the perspective of the local Firewall+ storage firewall, the deployment and registration steps constitute the provisioning transaction. To repeat, provisioning requires deployment then registration.

The SyncLink description (provided elsewhere) provides an overview of the normal provisioning process. In brief:

1. The Firewall+ storage firewall requests an update from the update server.

2. The update contains instructions, including that the Firewall+ should request an application

3. The Firewall+ requests & receives the application package

4. The Firewall+ installs the application package

5. The Firewall+ registers the application software

To repeat, provisioning the functionality provided by the application software requires deployment of the software, followed by registration into the application Rights table.

When new application software is deployed, the Firewall+ receives verification and authentication data with the downloaded package. This data is used to authenticate the received package. This is in addition to the authentication done during the SyncLink operation to authenticate each of the endpoints, the Firewall+ and update server, to each other. In addition, the package verification and authentication data is used to verify that the package has not been damaged or modified since leaving the update server. If the package can be successfully authenticated and verified, it may be deployed.

Software registration is part of the provisioning transaction. Registration modifies the application Rights table. Registration is done after the Firewall+ storage firewall update manager verifies the installation. Following the instructions in the package, after deployment the Firewall+ adds into the rights table the application runtime authentication information (application-signature) provided inside the downloaded package.

Either at this same time, or at some previous time, the same application-signature is (or was) provided to the application, so at runtime it will be able to authenticate itself to the Firewall+. In addition, the particular application-signature may have a different nature for different applications. The result of this flexibility is that some applications may be provisioned on a wide scale with common application-signatures, others are provided with unique application-signatures before being packaged for deployment, while others are provided with application-signatures generated locally by the Firewall+ storage firewall. [This and other mass-customization of the applications can be done on a wide scale, with the complexity handled by the intelligence governing the update server.]

The intent of the ability to apply different authentication schemes (or permutations of the basic authentication scheme) to different software applications is:

to support contractual obligations to software authors,

to support user customization (at the user portal web site)

There are multiple types of application provisioning, with some differences in how the Firewall+ conducts the local transaction. The two most common are

New software to be installed into the Bubble

Replacement of software currently installed in the Bubble

A related transaction is the removal of currently installed application software. This is not part of the provisioning transaction, but is mentioned here for completeness. When new application software is deployed to a Bubble, it is not available for use until it has been registered. At that point, a final step is required before it is provisioned for use. This final step is generally done during the registration process, but may be done separately, later. This step is to set the active flag in the appropriate entry in the Rights table.

The active flag can have several states, including fresh, active, inactive, and purge.

Newly deployed software is generally fresh before becoming active. When replaced, software generally becomes inactive as the replacement becomes active. The purge state indicates the software is about to be (or due to be) removed (deleted) from the Bubble and the Rights table. Therefore, it is possible for there to be as many as four extant versions of the same software model in the Bubble, but it is unlikely that there would be more than three. At no time will more than one version be flagged as active. In addition, only an active version of the software can be successfully authenticated to the Firewall+ storage firewall.

Firewall+ Startup

This section discloses the process whereby the Firewall+ Startup and Diagnostics (FSD) program transfers control to the Firewall+ storage firewall. In a software implementation, before the Firewall+ storage firewall starts running, the FSD program runs. This is also discussed elsewhere as part of the Firewall+ startup mechanism. This section provides an overview of the transfer of control from the FSD to the Firewall+ storage firewall.

The process by which the user authenticates himself (herself) to the FSD is covered in the authentication, authorization, and access control (AAA) section of this disclosure. It is a relevant topic because clearly the user has to authenticate at some point before the Firewall+ starts accepting application requests, and it does not happen during or after the FSD transfers control to the Firewall+ storage firewall.

The Firewall+ storage firewall can be implemented in several different ways. Chief among these is (1) as a software application with driver level hooks into the host operating system, and (2) as firmware within a storage controller. The transfer of control to the Firewall+ storage firewall is actually similar for both of these, but there are some differences, so a first option is discussed, then the differences with a second option are provided. Other options for implementation of the Firewall+ storage firewall have similar startup methods.

When the Firewall+ storage firewall is implemented as a software application with driver level hooks into the host operating system, the default startup process is:

The FSD does its startup and diagnostics actions, including examination of the Firewall+ storage firewall executable for evidence of damage or tampering. (This means that the FSD must have a way of opening the Bubble, since the Firewall+ storage firewall executable is within the Bubble.)

After validation, verification, and any repair, the FSD starts the Firewall+ executable. The actual transfer of control is done by sending the Firewall+ a Control transaction, then getting a 'success' response that matches a sanity value expected from this deployment of the Firewall+ storage firewall.

However, before transferring control (i.e. before the Control transaction), the FSD sends the Firewall+ an authentication and health (Health) transaction. If the response is not one of Health, the Firewall+ executable is terminated and the backup Firewall+ executable is started.

Then the Health transaction is re-attempted. If it succeeds, the Control transaction is used, then the parent process (the FSD) can exit.

If the Health transaction to the backup Firewall+ executable fails, the FSD kills the Firewall+ executable, then attempts to repair first the primary, then the backup Firewall+ storage firewall executable, using error correction data stored in the Bubble. If the repair succeeds, the Firewall+ can be started, followed by the Health transaction.

If neither of the executables can be repaired, the FSD will inform the update server to obtain a fresh Firewall+ executable. If the attempt to replace the Firewall+ executable fails (perhaps because there is no current SyncLink to the update server), then the FSD exits with an error message to the user; the Firewall+ can not be used at this time.

This default behavior can be modified by either a system implementer or by configuration changes. These modifications can be made to a single Firewall+ or to some defined set of Firewall+ protected devices.

When the Firewall+ storage firewall is implemented as firmware within a storage controller, there is similar startup behavior, but with some differences.

One major difference is due to the assumption that firmware in the controller is less likely to become damaged by malware or other system problems, so is less likely to need to be repaired or replaced by the FSD. As a result, there will not be a backup for the firmware Firewall+. It also may be more difficult or even impossible for the FSD to download and replace an 'unhealthy' firmware Firewall+. For example, if the Firewall+ firmware is part of an ASIC, this is not field re-programmable. On the other hand, an FPGA may be field re-programmable.

Firewall+ Initiation (Handshake)

This operation initiates communication with one or more currently running Firewall+ storage firewalls. There will be different implementations on different platforms, but essentially this sets up an inter-process communication channel, according to the mechanism used on that host computer's operating system.

When an application sends an initiation (handshake) request to the Firewall+ storage firewall, the values stored in the application rights table (Rights table) determine whether that application will be permitted to access the protected storage (Bubble).

The initiation transaction request should be added to the application's startup code. If we assume the application is written in C or C++, and the inclusion of the header file Firewall+.h, the initiation transaction request would look something like this:

F+_initiate(application_signature, &auth_token);

where F+_initiate( ) is a function provided by the F+ API, and the application_signature is a set of cryptographic information that identifies and verifies the identity of the application executable. The application_signature was provided to the application at some previous time, perhaps by linking it in before it was provisioned. Other information on the application_signature and authentication token can be found in the section on authentication.

Request/response protocol, even though the application should not see most of these, as they will be hidden within the ABI

There are no changes to the Rights table from this transaction, but this authentication request may be logged.

Application File Access

Open/Read

From the perspective of the typical application, the Firewall+ file-open and file-read transactions should seem to have the same syntax and semantics as the standard operations. The Firewall+ ABI handles the access control issues, adding the authentication token to each transaction.

There are no code changes to the application for read access to a Bubble-protected file, other than for the Firewall+ initiation.

Open/Write

From the perspective of the typical application, the Firewall+ file-open and file-write transactions have similar syntax, but there is a significant difference in the behavior: there can be only one application that has a given file open for write at any time. Some operating systems permit multiple applications to have the same file open for write; the Firewall+ storage firewall does not permit this.

The restriction against multiple writers can be eliminated by providing the same sort of inter-process communication (IPC) mechanisms and/or file locking as many operating systems. These were avoided to simplify the Firewall+, as well as to reduce its footprint (size), so it can fit into smaller devices, perhaps as part of an ASIC. This restriction is not likely to have much affect on most of the applications that will be encountered.

The Firewall+ ABI handles the access control issues, adding the authentication token to each transaction.

For many applications, there need not be code changes for write access to a Bubble-protected file, other than for the Firewall+ initiation. Of course, in practice, this depends on the application software design, etc.

Extensibility

Both the SyncLink and Firewall+ transaction models have been designed for maximum extensibility. This is an important quality of the Firewall+ architecture, no matter what implementation choices are made. Some of the extensibility options are:

Transactions can be added,

Scripts can be sent to the Firewall+ storage firewall for local execution,

Hidden application software can be selectively deployed

Firewall+ storage firewalls can be replaced by later versions

Augmenting the System by Adding New Transactions

An implementer can add a new transaction or transaction type very easily. An existing system of deployed & provisioned Firewall+ enabled devices can also be augmented, by adding the desired transaction infrastructure, then replacing the Firewall+ storage firewalls in the field through the update mechanism.

Firewall+ Command Execution: Scripts

The update server may send to the local Firewall+ storage firewall arbitrary scripts to execute. The Firewall+ has the ability to execute these scripts. The scripts have the same high level of authentication that other SyncLink downloaded packages have, so the intent of these scripts is not questioned by the local Firewall+. These scripts are authenticated and verified in the same way as application packages, but are not added to the Rights table, because they are generally executed just once, soon after download or at a loosely scheduled time. This functionality is generally used for maintenance operations, but may be used for many other activities. I

Scripts are generally transient, provisioned when needed. If a more permanent tool is needed, a system tool is deployed.

Examples of use are:

the cleanup (purge) of old versions of software littering the Bubble,

backup or restore operations

quality of service monitoring, usually after the authorized customer reports a problem

disabling a provisioned service after the license expires

tracking the location of a device that's been reported as stolen by the authenticated owner.

Hidden Software

The update server can selectively deploy hidden applications to one or more Firewall+ storage firewall protected devices. This software is deployed & registered in the normal way, but will not be evident to the user. In the preferred embodiment, the application-browser will not normally show the user the hidden applications deployed to the Bubble, but the application-browser can be adjusted by the user through an application configuration mechanism.

A category (type) of hidden software is system software. Most hidden software is system software.

System software applications are generally for maintenance, security, system management, quality-of-service monitoring, or other background responsibilities. They can be (may be) deployed without being activated, so they will be in place if needed later.

Augmenting the System Through Field Replacement of Firewall+

The Firewall+ storage firewall can be field replaced in the same manner as any other deployed software. The new package is deployed, then provisioned. The next time the Firewall+ storage firewall is started, the latest version is used.

When a new version of any software is provided, the most recent (proven stable) older version remains. The earlier version is not deleted until some time later, when the update server sends an instruction to do so, generally when there are two older versions in the local space.

If the newly provisioned version of the Firewall+ storage firewall won't start properly, the Firewall+ Startup and Diagnostics program will automatically kill the latest and start the earlier (proven stable) version.

Status information is automatically provided to the update server from the local Firewall+ storage firewall on each update request.

The Firewall+ Storage Firewall as Server

The Firewall+ Storage Firewall and update manager system has been designed to be extensible, as described elsewhere in this document. A related design goal is to use this extensibility to selectively enable Firewall+ Storage Firewall protected devices as relay servers to other Firewall+ Storage Firewall protected devices. There are many benefits to this design, not covered in this document.

Peer-to-Peer (P2P) Communication

The secure communication channel between the Firewall+ Storage Firewall and it's designated update manager servers can be directed to other Firewall+ Storage Firewall protected devices, and the transaction set on those other devices can be augmented with update manager services. This effectively defines those other devices as servers. This scheme can be extended, so that every Firewall+ Storage Firewall is both a storage firewall for local protected storage, and an application and general purpose update manager for other (peer) Firewall+ Storage Firewall protected devices.

The same SyncLink authentication mechanism defined for Firewall+ and Update Server authentication can be used for P2P SyncLink endpoint authentication. In addition, each Firewall+ Storage Firewall SyncLink module will have a flexible list of Update Managers to query for designated updates. This list can contain peers acting as servers as well as update servers that are not peers.

In addition, the Firewall+ SyncLink module can use a discovery mechanism to search for Firewall+ update servers and relays. This is covered in more depth in the SyncLink design document.

Pervasive Computing Vision

The purpose of using Firewall+ Storage Firewall protected devices in a P2P collaborative network is to map around possible network server outages, reduce Update Server bandwidth and server CPU congestion, and in general to provide a faster, more reliable, and better user experience.

Another use of the P2P nature of SyncLink communication and the application server nature of Firewall+ Storage Firewall enabled devices is that applications on multiple such devices can collaborate on tasks. In addition, applications can move between such devices, obtain more compute resources as needed, and do other collaborative actions in the pursuit of their goals. More on this aspect of the Firewall+ architecture in the application notes document.

Firewall+ Storage Firewall as Application Server To Local Host

The Firewall+ Storage Firewall, even if not in a P2P collaborative, essentially functions as an application and storage server to the local host. This is because of the transactional nature of Firewall+ Storage Firewall storage protective protocol. This differs from the typical local (i.e. direct attached) storage protocol which are often queue based, designed to minimize CPU involvement and cycle cost.

Authentication, Authorization, and Access Control

The term Authentication. Authorization, and Access Control is usually abbreviated as AAA. Some definitions of AAA use accounting instead of access control—in the Firewall+ design it is access control.

There is some transaction specific detailed information on AAA provided in the Firewall+ transaction sections—this part of this document provides an overview of the authentication architecture, plus additional details. If there seems to be a contradiction in the details, this AAA section is correct on the AAA approach, architecture, and general mechanism.

Authentication, authorization, and access control are intimately related.

In terms of the Firewall+ storage firewall design, these terms are used in this way:

- Authentication is the identification step—it decides whether an entity (user or application software) is recognized. It does not seek to prove the identity in some real world manner, merely to verify to some degree of assurance that the entity is the same as is known.
- Authorization uses a lookup of rights and privileges to decide whether an entity can be permitted to do some action or some set of actions.
- Access control uses the authentication and authorization information to decide whether to permit the entity to a specific action at a specific time (where the time is typically at the moment the action permission is requested.)

In most designs, the authentication and authorization operations are provided by the same mechanism. In the Firewall+ architecture, this is also the case, but the default AAA algorithms are designed such that these can be adjusted independently.

In addition, there are two different yet related AAA models used by the Firewall+ storage firewall. One is for the application software interaction with the Firewall+ storage firewall, while the other is for the user (or some other software entity that represents the user) to "login", so the user can then be permitted to request that a Bubble protected application be started. From these two models are derived two sets of authentication mechanisms, one for the user and the other for software that intends to access the Bubble. There is a link between them, but basically they function independently. The user authentication design is simpler, so it is handled first.

Referring again to the diagram shown in FIG. 3, the key data structures and data elements are put in context to each other. These are described & explained below.

User Entity AAA

This section provides details on the way the user (device owner, or some software agent representing the user) interacts with the Firewall+ storage firewall protected device.

There is an assumption in these descriptions that the device is a USB flash drive (UFD) being used on a personal computer, perhaps one whose operating system is MS Windows XP. This interaction model is transferable and extensible to other operating systems, other platforms, and other types of protectable devices and storage media.

There is also an assumption of plug-in authentication (or plug-in AAA). In particular, while the default model is very simple, the mechanism can be easily replaced by third party user authentication. The replacement might have a more elaborate design and/or biometric mechanism, or some other innovation in identifying the user.

There needs to be a way for the user, typically a human, to authenticate himself (herself) to the Firewall+ storage firewall. In other systems, user IDs and password combinations are typical. This is also the default model used by the Firewall+ architecture.

When a user obtains a new Firewall+ protected device, such as a USB flash drive, the user has to select a user ID and password. The user interface method is not described here. The user ID and password are stored on the new device, among other data that collectively are referred to as User Registration Data.

The User Registration Data is stored in a special place inside the Bubble.

Later, when the user wants to use the device, the same user ID and password must be entered, compared to stored values in the User Registration Data.

When the user ID and password, along with possibly other data, are entered, there is another field created in the User Registration Data. This is the independent user credentials data field. The user credentials data field is also provided to the update manager; it is used as part of that user's identity to the update server. As such, it becomes part of the licensing and/or authentication mechanism for some software applications. There may be links from some deployed application software to the User Registration Data's user credentials data field, for license verification and/or user authentication at application startup.

Software Entity AAA

This section covers all application and other software that needs read/write access to the Bubble (protected storage). This class of entity is referred to as the application or application software in this section. It might include other software such as the host computer's operating system, in some implementations.

Plug-in Authentication

The Firewall+ storage firewall architecture permits a plug-in model for AAA modules. This permits it to leverage industry standards for authentication and authorization. For different implementations, an implementer can select the third party authentication module that best meets the specific requirements. The authentication module has to be compatible with the Firewall+ architecture, but on a survey of existing available third party authentication modules, many were compatible.

The exact method used in a particular implementation will depend on which corresponding AAA library was used. The default AAA architecture is described here.

Authentication Token and Credentials Table

During the Firewall+ initiation (handshaking) transaction, application authentication and authorization take place, and the application is supplied with an authentication token (auth_token). Accompanying subsequent file access requests, the authentication token provides a sort of permission ticket for the application.

The auth_token authentication token is a hash of several values including an index to the credentials stored in the Credentials table. The index can be recovered by mapping out the other values.

The Credentials table is transient, created as needed by the Firewall+ storage firewall, generally as part of the reaction to an initiation transaction request. It stores a version of application authentication, authorization, and access credentials and rights.

In some implementations, the only reason to create and store the Credentials data is when an auth_token is created. In other implementations there will be Firewall+ internal mechanisms that also make use of this data structure.

In the simplest implementation, with only one possible application, the Credentials table will have only one data set.

The auth_token index is provided to the application, but the contents of the other Credentials table's data fields are not exposed to the user nor to the application. The Credentials data includes an index to the Rights table.

Default Authentication Module

The software AAA default authentication module uses an application_signature mechanism. This can be thought of as a password, but longer. The exact application_signature mechanism will vary across different software applications, based on the licensing requirements imposed by the software author. The key thing is the Firewall+ and the application to agree on the application's application_signature.

One application_signature scheme is for it to be the hash of a shared secret and some obtainable information. The obtainable information might include the process id (PID) of the application, a value for the current time and date, and a value for the date the application was installed to that device (i.e. the Bubble on that host computer system).

The application passes the application_signature to the Firewall+ storage firewall with the initiation request.

The application_signature, or a way to generate it, is provided to the Firewall+ Rights table by the update server.

Corresponding information is provided to the application either before, during, or even after the application is deployed. It is expected that the same application software on different deployments will use different shared secrets.

This default authentication module is easily extensible. Extensions might be additional information added to the hash, or differing hash algorithms used by different software applications.

Authentication Module Version Number

The initiation request (transaction) uses an authentication module version number to indicate which authentication module and/or which version of the authentication's algorithm is being used by that transaction. It is assumed that the Firewall+ storage firewall knows how to dynamically select the correct algorithm based on this version information. This assumption will only be correct if this Firewall+ storage firewall has received an appropriate code module from the update server.

Authorization Per the Rights Table

Before the application requests access, the application's access privileges (or rights) are stored in the Rights table. These are provided by the update server.

If the application will be resident within the Bubble, the Rights table generally receives authorization information when the corresponding application is provisioned (deployed) to that device or Bubble.

The application rights data stored in the Rights table represents not merely the generic application's rights to file access—this data (and the corresponding access privileges) is formed from the combination of the application's generic rights (and configuration) plus the user's rights and privileges. The update server combines these. The update sever can also be used to modify these. The updated Rights configuration is provided to the local Firewall+ in the normal update manner.

When the application sends the initiation request, the Firewall+ storage firewall uses the authorization information, to form the decision tree for subsequent authorization processing. If some level of access is to be permitted, based on a combination of application Rights and User Credentials, then an appropriate entry is made to the Credentials table, and an authentication token is returned to the application.

The authentication token, auth_token, is a hash of validating data and an index to the appropriate entry in the Credentials table. There is a link from this entry to the application's entry in the Rights table.

Access Control

After an application has been authenticated and authorized to access the Bubble, the application receives an authentication token (auth_token). This is sort of like an access ticket. It is a reference to the appropriate entry in the Credentials table.

The authentication token encapsulates or represents the application's rights to the Firewall+ storage firewall that generated it.

The interesting thing about this scheme for access control, is that the application's runtime access rights (i.e. the data in the corresponding Credentials table entry) can be changed while the corresponding authentication token is extent.

Transaction State Table

This section provides an overview of core transactions. Core transactions are those most frequently implemented, suitable for the widest set of implementations.

The transaction state table represents the request-state-action-response path of transactions. This may or may not be implemented by an actual state table in the Firewall+ storage firewall software. The transaction state table is presented in this document as a representation of the decision tree of the Firewall+ software. Other implementations of the decision tree, in addition to a state table, could be a case statement, or a set of If-Then statements, or some other language and implementation-specific option.

The way to read the state table documentation is:

when a request is received,

if the current state is as shown,

then the corresponding action takes place,

followed by the corresponding response to the requester.

In addition, this state table documentation has a column for Flags; these provide additional information such as indicating the transaction class. These Flags are optional, and only sometimes accurate; they are provided as guides to the implementer. But in fact, all of the contents of the documented state table can be viewed as non-authoritative—the actual transactions in an implementation can be completely different. The key design element is that the Firewall+ architecture is transactional, not that there are specific transactions.

In this disclosure there are classes of requests (transactions). These class designations are for ease of design, development, and test. Depending on the implementation, there may not be a functional impact of these transaction classes. Some of the transaction classes are:

From the update server, embedded in or as part of SyncLink transaction responses;

From application software

Generated by the Firewall+ storage firewall mechanism(s) itself

For ease of design, development, and test, there may be different sets of request (transaction) handling and authentication processing for each of these.

Some of these transaction classes are implemented only in a software version of the architecture, others only in a Firewall+ Storage Firewall that is capable of using the SyncLink module to reach an Update Manager server.

Flag Key

Currently, only transaction class flags are provided. These are: U

from an update server

A—from an application or other software (non-update manager, non-Firewall+ Storage Firewall)

F—from the Firewall+ Storage Firewall itself

For example, provision, update, and arbitrary-execution requests (transactions) are from the update server. These arrive by secure SyncLink. These transactions are verified (authenticated) by SyncLink methods.

TRANSACTION TABLE

Flags	Transaction	Current state	Action	Response
U	Provision Transaction	Application not registered	Application is installed, Application is registered in the Rights table	Success if successful; otherwise the appropriate error code
U	Provision Transaction	Application already registered in the Rights table for access to this Bubble	New application is installed, old application is tagged as ‘replaced’; Registration info in the Rights table is updated in parallel. On the next execution, the newer version will be used.	Success if successful; otherwise the appropriate error code
U	Update Transaction	Application not registered	None	Error code sent back to the update server over the SyncLink channel
U	Update Transaction	Application already registered in the Rights table for access to this Bubble	New application is installed, old application is tagged as ‘replaced’; Registration info in the Rights table is updated in parallel. On the next execution, the newer version will be used.	Success if successful; otherwise the appropriate error code
U	Execution Transaction	Referred-to-Package not present or not healthy	None	Error code sent back to the update server over the SyncLink channel
U	Execution Transaction	Referred-to-Package previously received, present, healthy	Script instructions in the Package are executed if possible And reasonable (local discretion possible). Status result from the script is prepared for response	Success code and prepared script response are sent back to the update server over the SyncLink channel
A	Initiation request	Application not registered	none	Error code
A	Initiation request	Application registered for access to this Bubble	Entry added to credentials table, etc. authentication token generated	Authentication token passed to application for use in access calls
A	Request to open a file for write	Authentication token is not valid	none	Error code
A	Request to open a file for write	Authentication token is valid	File is already open for write, so this request must fail.	Error code
A	Request to open a file for write	Authentication token is valid	File is opened for write, which means an appropriate entry in the file table, etc.	Success code, file-reference (fp) is provided
A	Request to open a file for read	Authentication token is not valid	none	Error code
A	Request to open a file for read	Authentication token is valid	File is opened for read, which means an appropriate entry in the file table, etc.	Success code, file-reference (fp) is provided
A	Write access request	Authentication token is not valid, or file reference is not valid for request type	none	Error code

-continued

TRANSACTION TABLE

Flags	Transaction	Current state	Action	Response
A	Write access request	Authentication token is valid, and file reference is valid for request type	Data written to file	Success code
A	Read access request	Authentication token not valid, or file reference not valid for request type	none	Error code
A	Read access request	Authentication token is valid, and file reference is valid for request type	Data read from file	Success code, read file data
F	Health	Firewall+ Storage Firewall not healthy	Response on non-health condition prepared as response	Success code, prepared health response

Functional Descriptions

Referring now to FIG. **5** of the drawing, a Remote Management System in accordance with the present invention and including a Storage Firewall Storage Device **100** as described above is shown. The system is composed of elements of a Configuration Web Server **200**, an Update Server **202**, a Host (Host Computer) **204**, and the Storage Device **100** including the Host Interface **10**, Storage Firewall **12** and Protected Storage **14**.

The Internet component, as shown at **206**, is not strictly speaking a component of the Remote Management System, but it is assumed to exist and be present for the System to function.

The Configuration Web Server **200** enables end users and owners to select customization options for their Storage Firewall protected storage and endpoint devices.

Update Server **202** is the remote management system's update server, providing updates, configuration changes, new software, and other information and data bundled into discrete Packages PB. The Update Server, as shown, contains a Configuration Database module **210** and a SyncLink module **212**. The decision as to what information, data, software, and files to bundle into specific Packages PB depends on the information in the Configuration Database and a set of algorithms not detailed here.

The Configuration Database **210** contains configuration information about each Storage Firewall protected storage and endpoint device **100** (hereinafter called device), where such configuration information is a combination from several sources:

Class A: Generic

configuration information derived from the device's serial number such as model, version, etc., combined with history information such as the most recent previous configuration update successfully provisioned, and Class B: user customizations

At any previous time, including just previous to an update request, the owner may have selected customizations for this device's configuration. and Class C: derived from device status information combined with information provided by the device regarding its operating environment, the network bandwidth for the connection, etc.

The SyncLink module **212** provides an intelligent communication channel from the Update Server to the distributed Storage Firewall protected storage and endpoint devices **100**.

The Host (Host Computer) **204** is the Host that the Storage Firewall and its protected storage is currently attached to. As described above, the Host Interface **10** of a local device **100** and provides an interface between the Host and the Storage Firewall **12**.

For simplicity and the purpose of this discussion, the Storage Firewall **12** is shown to include only the SyncLink module **34**, the Application Rights & Credentials module **26** and the State Engine module **28**.

As described above, SyncLink module **34** provides an intelligent communication channel to the remote management system's update server **202**, while the Application Rights & Credentials module **26** stores, processes, and provides user and application credentials, and access parameters for authentication, authorization, and access control purposes.

The State Engine module **28** provides the execution of the transaction decision tree or state table of the Storage Firewall. All transactions are processed through the State Engine module, which decides what to do, how to respond, and which other modules and components to use to meet the requirements of the requested transaction.

The Protected Storage **14** is the storage resource being protected by the Storage Firewall **12**.

There is a multi-part data path that can be followed, through the several illustrated operative parts of the system connecting the Configuration Web Server **200** to the Protected Storage **14**. This path is completed by a data path d**1** which connects the Configuration Web Server to the Update Server's Configuration Database **210**; a data path d**7** which connects the Configuration Database to the SyncLink module **212** (The data path d**7** actually represents several modules of the Update Server which are not shown.); a data path d**2** which connects the Update Server's module **212** to the Internet **206**; a data path d**3** which connects the Internet to the Host **204**; a data path d**4** which connects the Host to the Host Interface **10**; a

data path d**5** which connects the Host Interface to the Storage Firewall **12**; and a

data path d**6** which connects the Storage Firewall to the Protected Storage **14**.

Internal to the Storage Firewall **12** is a control connection and data path c**1** between the SyncLink module **34** and the State Engine module **28**, and a control connection and data path c**2** between the State Engine module **28** and the Application Rights & Credentials module **26**.

In the illustrated diagram, and within the Update Server **202**, a Package PB representing Packages formed within the Update Server is shown. Similarly, within the Storage Firewall **12** a Package PA, representing Packages received by the Storage Firewall, is depicted.

The Configuration Web Server **200** can be implemented using any reasonable set of web server tools, according to the navigation design provided in the Firewall+ Architecture described herein. One choice for the Configuration Web Server technologies and tools might be the Apache web server with Java backend, JavaScript coded web pages, and firewalled network integration with the Update Server's Configuration Database.

The Update Server can be implemented using any suitable software-server tool set. An example is the Apache web server with XML, SOAP, database-integration, bit-torrent (peer-to-peer, P2P) integration, and related modules integrated for data processing performance.

The design of the SyncLink protocol drives many of the implementation requirements of the Update Server. In particular, the requirement to serve up (deploy) Packages targeted at a few or one specific supported device, while also serving up Packages intended for millions of supported devices requires the support of ad-hoc P2P collaborative peer networks, based on the intended recipients of specific update Packages. While this does not change the basic architecture, it does add complexity to the design.

The Configuration Database can be implemented using any of multiple relational database designs. The basic requirements for the Configuration Database implementation are a query engine and a storage manager. Other commercially available relational database features are used in the implementation of the Configuration Database in order to improve performance and security. These include query language, views, triggers, and symbols. There are several suitable available products and technologies that can be used to provide these.

The SyncLink module **34** provides an intelligent communication medium for providing requested updates to supported devices. It is transactional, but with "fuzzy" delivery, scheduling, and packaging of the actual updates. It is designed around 2 sets of interoperable IP-based protocols.

The first set of protocols is commonly thought of as web protocols: HTTP, HTTPS, TLS (SSL), XML, SOAP and others that have been applied and/or extended to provide and support a portion of the SyncLink protocol.

The second set is derived from the bit-torrent or BitTorrent protocol (really a family of protocols). When the SyncLink module transfers a get update request to the Update Server, the Update Server attempts to reply to the requesting device as quickly as possible using the SyncLink module, using the part of the SyncLink protocol based on web-based protocols. The reply to the requesting device is a Package that may necessitate additional update requests or that may instruct the device to obtain portions of the update from other peer devices. There is flexibility in this process, directed by an intelligent module (not shown) of the Update Server. The SyncLink module's implementation permits similar requests from different devices to be replied to quite differently, in an adaptive manner, based on knowledge of network load, server load, network topology, and other factors.

The Internet component is provided by the Internet and is interfaced to by standard network equipment (not shown).

The Host (Host Computer) **204** can be a PC or any other computer, computing device, digital equipment, or analog equipment, with interfaces to the Internet and to the Storage Firewall protected storage.

When embodied in a USB flash drive, the Host Interface **10** is a USB controller. As suggested above, USB controller chips and designs are commercially available from manufacturers such as Cypress, Anchorchips, Scanlogic, and Intel.

The Storage Firewall **12** filters access from the Host to the Protected Storage component and has three relevant subsystems including the SyncLink module **34**, the Application Rights & Credentials module **26** and the State Engine module **28**.

The SyncLink module **34** is controlled by the State Engine module **34** which periodically uses the SyncLink module to poll a remote management system's update server for updates. When an update is received, the SyncLink module opens up the retrieved update Package PA and provides the contents of the update Package to the State Engine module for processing.

The Application Rights & Credentials module **26** is implemented as a set of data structures and operations that act on data structures which

include the Application Rights Table, the Credentials Table, and the User Registration Data depicted in FIG. **3** above.

The Application Rights & Credentials module is able to store new and updated information in these data structures, and provide functionality (object functions visible to and usable by the State Engine module) to manipulate, adjust, and update the contents of it's data structures.

The State Engine module **28** is the transactional portion of the Transaction Processor **20** described above. Implementation of this module is based on a software coded state table, an implementation of the Firewall+ Storage Firewall decision tree. To save space on the chip, the decision tree is encoded in a state table. The state table has an entry for every transaction provided by the Firewall+ Storage Firewall. If the chip is an FPGA (Field Programmable Gate Array) or other field-changeable chip or chip portion, it may be possible to upgrade or replace the state table in an operational Storage Firewall's Transaction Processor component's State Engine module. This is useful to add transactions, or to improve or change the behavior of existing transactions. It can also be a security vulnerability, if an attacker is able to change transaction behavior. However, this implementation is preferred in some consumer products, to reduce product support costs.

If the chip is an ASIC (Application-Specific Integrated Circuit) or other non-field-changeable chip or chip portion, then it is not possible to upgrade or replace the state table in the field. This implementation is preferred in a highly secure product or installation, such as a military application.

The Protected Storage component **14** may be implemented as NAND Flash chips.

Major transactions are described in relation to the functional diagram. These transactions include

Request and receive update for an existing application, where this update provides a modified application configuration and application rights.

The provision of basic status information by the Storage Firewall to the Update Server, prompting a more complete status report from the Storage Firewall to the Update Server.

Transaction: Request and Receive Update, for Application

This is a client-server transaction from the Storage Firewall to the Update Server.

Periodically (asynchronously) the Storage Firewall polls the Update Server for updates. There is no fixed schedule for this poll, since it is impossible to predict when or how often the Storage Firewall protected storage and/or endpoint device will be in use.

The update transaction is independent of whether the Configuration Web Server has been used to modify the records for this Storage Firewall in the Configuration Database **210**.

When the Storage Firewall sends an update request transaction to an Update Server, the SyncLink module **34** selects which Update Server to contact, maintaining information that supports such contact. The update request transaction goes from the Storage Firewall's SyncLink module through the Host Interface to the Host (Host Computer), where it is packetized by the Host's network interface (not shown) to be suitable for transmission over the Internet, and sent through the Internet to the Update Server.

On the Update Server, the SyncLink module **212** receives and validates the update request transaction, interprets the update request transaction, and specifies the transaction response. The SyncLink module then packages the transaction response, the update, shown as Package PB, and directs it back to the Storage Firewall, over the Internet. (Not shown is the Update Server's network interface.) The update (the transaction response) is received by the Host **204**, and passed through the Host Interface **10** to the Storage Firewall **12**.

When the update is received by the Storage Firewall, it may include several elements. One of these is a Manifest (not shown), containing a list of other elements, including a software Package PA. The SyncLink module **34** opens the update, and provides the Manifest and Package PA to the State Engine module **28**. The Manifest provides the State Engine with instructions, including what to do with the Package.

In this transaction example, the Package PA might contain changes to the configuration of an Application G1 stored in Protected Storage **14**, as well as corresponding changes to the application rights for this Application stored in the Application Rights & Credentials module **26**. The Application's configuration changes are applied by the State Engine directly to the Application by writing the received (updated) configuration file into the appropriate place within the Protected Storage. The changes to the application rights for this Application are effected by the State Engine module using the Application Rights & Credentials module **26**.

Transaction Set: Basic status information to the Update Server, instruction from the Update Server to send complete status report,

There is no acknowledgement message sent from the Storage Firewall **12** to the Update Server for completed update transactions, but periodically (asynchronously) the Storage Firewall sends to the Update Server basic information regarding it's status, including information on it's current configuration. This is called a Status Poll. In this way, the Update Server's Configuration Database **210** is kept up-to-date on the health, configuration, etc. of this Storage Firewall **12**.

If the If the Update Server **202** is satisfied with the status information received in the Status Poll, the response will be either a basically empty acknowledgement (or some other response that elicits an Update Server directed request from the Storage Firewall **12**).

If the Update Server notes a significant discrepancy from its configuration record on this Storage Firewall, or for some other reason, its reply to the Storage Firewall **12** will elicit a more complete status report. This called a Health Report, but note that it is still presented by the Storage Firewall as a request to the Update Server. The response from the Update Server to the Storage Firewall for a Health Report is formed and treated in roughly the same way as a Status Poll.

The Update Server's response to a Health Report may be intended to elicit an update request (as documented above).

The illustrated causal chain of requests and responses is diagramed in FIG. **6**.

In FIG. **6**, the Status Poll (1) is generated and sent from the Storage Firewall's State Engine **28** to the Update Server.

The Status Response (2) from the Update Server **202** contains an instruction to the Storage Firewall's State Engine that elicits the Health Report request (3).

The Health transaction's response (4) contains an instruction to the State Engine that elicits the Update Request (5).

The Update Response contains the actual Update Package applied by the Storage Firewall's State Engine.

For each transaction in this chain of requests and responses, the Configuration Database **210** is provided with information from the most recent request, then information from the Configuration Database is used by the Update Server **202** to decide what instructions, if any, to put into that request's response.

Scheduling Mechanism

The storage firewall has a scheduling mechanism or component.

The storage firewall scheduling component is used for several purposes. These include scheduling the polling (of the update server), scheduled execution of software deployed to the storage firewall, scans of the Bubble (protected storage) for anomalies (and other targets), etc.

The scheduling component has the requirement to balance the prioritization of read-write access and related top priority activities, versus secondary activities such as detecting and analyzing anomalies in the access request stream (followed by sending Warning messages to the update server), versus tertiary activities such as normal update server polling, software management, and disk maintenance actions, versus background activities such as disk scanning (malware, arbitrary strings, etc.). There can not be a single scheduler design that will fit all of the multiple possible designs of the storage firewall, but the following design of the scheduling component can be used as an example.

The design of the storage firewall scheduling component can be as simple as a list of "processes", each with a priority value, perhaps on a scale of 0 to 100, where 100 is the highest possible priority on this scale. A priority of 0 implies the process is currently "sleeping". These priority values are segmented into sets, for example 1 to 20 are background tasks. In addition, a 'real-time' "interrupt" scheme causes certain (high priority) processes to be executed as needed.

The basis of one design of the storage firewall scheduling component is a queue with priority interrupts, with several subsets based on priority. A version of the schedule component could be described as follows: processes and threads are added into the queue based on their priority, the queue is processed from highest to lowest (i.e. top to bottom, or next to last). There are preemptive interrupts for certain threads, based on their being among the highest priority set.

The storage firewall scheduling component requires a timing signal from a system clock. System clock generator parts are commonly used and available, or an existing system clock (i.e. the clock of the protected digital device) can be used. If the storage firewall requires a new clock mechanism, then either a PLL (Phase Locked Loop) synthesizer or an XO (crystal oscillator) module can be used, depending on the design of the embodiment. In any case, the system clock source provides input to the storage firewall scheduling mechanism.

If the storage firewall is designed so as to be integrated into a disk controller, the design of the storage firewall scheduling component will have to be modified as necessary. For example, if the storage firewall is integrated into an SSD (Solid State Disk) which is (itself) composed of DRAM (Dynamic Random Access Memory) and Flash memory, then the storage firewall's components will likely be split among the SSD controller, as well as the separate controllers for the DRAM and Flash. This means that the storage firewall can take advantage of aspects of the SSD design such as the DRAM refresh cycle and Flash idle time to perform disk maintenance activities and run scans of disk contents.

Disk Architecture Attributes Relevant to Storage Firewall Functionality

HDD controllers have a processor, often a bit of RAM for their own processing, but often also some RAM as a cache—HDD controller design does not matter from the perspective of this level of description of storage firewall functionality and design.

SSD architecture can be some type of RAM (usually DRAM) and/or Flash, with an overall SSD controller, plus each of the RAM and Flash will have their own controller.

Flash cells can be NOR or NAND, NAND MLC or NAND SLC, or some other technology. The type does not matter.

Flash solid state memory is a type of EEPROM (electrically-erasable programmable read-only memory), but with a significant difference that it is faster because it is read in blocks, not bytes as older EEPROM is. NOR Flash is a better replacement for EEPROM. NAND is better as a replacement for HDDs and CD-ROMs, etc. MLC (multilevel cells) is denser than SLC (single level cells), because multiple voltage levels can be stored per cell.

The flash controller (usually idle) can scan cells at no-performance cost to the system it is part of.

The storage firewall can use the DRAM refresh cycle for scanning the DRAM.

Other storage firewall functionality may be added to the SSD controller, depending on implementation, design parameters, market requirements.

The storage firewall can execute local software (operations) such as scans for malware hiding on the disk. In spite of the storage firewall's execution-whitelist and access-whitelist mechanisms, malware can still propagate to the disk, perhaps as part of files saved by a user when browsing the web (World Wide Web, or WWW). The current state of the industry (Prior Art) is for malware scan software to execute on the mission processor (i.e. the Central Processing Unit, or CPU). It will be much more efficient for these malware scans, and other scans, and various other software executables used in the maintenance of the disk and system to execute as part of the storage firewall. Since the storage firewall functionality is likely to be integrated in the disk controller, this means that the system maintenance tools will, likewise, be integrated into the disk controller.

The base storage firewall functionality is built around read-write access control and execution control based on a whitelist of software executables, plus the server architecture and SyncLink communication mechanism. The storage firewall processor can also, obviously, be used as a base for non-access-control features: including technology hooks that provide a significant amount of value: arbitrary executables, scanning the disk for arbitrary strings, built-in executables such as A-V scan, an awareness of what the disk controller is doing (ex. In an SSD whether the SSD is in a DRAM refresh cycle or whether the Flash controller is busy), and an overall scheduling mechanism (which can be used for a variety of purposes). Whether or not the storage firewall is integrated into the disk controller, it can scan on the disk (storage) for:

Arbitrary content, given a list of wildcarded strings (fuzzy search)

Unlicensed software

Unpatched software

Bad blocks and other disk issues (either hard drive or solid state disk)

Malware (for example, but not only, by looking for patterns that correspond to one or more parts of a 'black list' signature file)

Note: malware can end up on drive if written by legitimate (authorized) software such as an email client application saving an infected attachment. Scanning for it and removing it should be done by storage firewall rather than an application on the CPU because the entire system can run faster when tasks are off-loaded from the CPU.

Typical storage firewall scan (for whatever reason) will occur when the hard drive is less busy. On some SSDs, those that employ DRAM, the scan can occur during the DRAM 'refresh cycle'. Other storage firewall services may include those that act upon the disk (hard drive or SSD, where appropriate), such as:

defragment hard drive

balance flat file databases' data structure and other database maintenance operations

secure delete of drive contents

provision arbitrary software, including system software modules

delete, remove, manage licenses, such as software licenses

These scanning and other executable operations, such as malware scanning may be provisioned from the update server, or they may be hard coded (built into the storage firewall) or predeployed, perhaps when the storage firewall or the enclosing protected device is manufactured.

Storage Firewall Functionality Integrated into Solid State Drive's (SSDs),

There are several designs for SSDs. One of these uses DRAM in combination with flash, where the design may use NAND MLC flash. The goal is to provide a very fast drive for swap space and other purposes, while using the flash as a permanent store when the drive is not powered. Flash has a relatively slow write speed, and the power to the drive may be cut at any time (ex. power failure), so the drive has to have a mechanism that can trigger the contents of the DRAM to be written to the flash component. (The drive also has to have an independent power supply to ensure that there is enough power during the time it takes to transfer this data.)

FIG. 7A is an illustration of the Prior Art, an SSD (Solid State Disk) without Storage Firewall functionality.

As illustrated in FIG. 7B, the I/O Bus (S1) connects the SSD to the rest of the digital system of which it is part, and connects to SSD Component (S2), the SSD Controller. The SSD Controller may or may not have Storage Firewall functionality, without prejudicing whether other components of the SSD have such functionality.

Further, in FIG. 7B, the SSD Controller (S2) connects to the DRAM Controller (S4) through local memory bus (S3). The DRAM Controller may or may not have Storage Firewall functionality, without prejudicing whether other components of the SSD have such functionality.

Further, in FIG. 7B, the SSD Controller (S2) connects to the Flash Controller (S8) through local memory bus (S7). The Flash Controller may or may not have Storage Firewall functionality, without prejudicing whether other components of the SSD have such functionality.

The DRAM Controller (S4) connects to the DRAM (S6) through local memory bus (S5).

The Flash Controller (S8) connects to the Flash (S10) through the parallel array of local memory buses (S9).

There are use cases that require a software embodiment of the F+ Storage Firewall. These include, but are not limited to, any computing and information access environments that do not have an integrated hardware embodiment of the F+ Storage Firewall. These include, but are not limited to:

Storage controllers that support add-on software

Virtual computing environments

Cloud computing environments

The present invention provides a software embodiment of the F+ Storage Firewall and Support Server technology to provide security functionality integrated into a customizable storage controller, which is a replaceable software storage controller, or in other words, software executables that implement and provide storage controller functionality.

My earlier provisional patent application referred to above described a tight bundling of application software and storage firewall, to be deployed on a USB Flash Drive, as a FlashApp. The provisional defined a software embodiment that was intended for the support and protection of single application software executables and small sets of related software executables. The stated purpose was the protection of the application executable and it's associated data. In particular, the stated purpose was the protection of the integrity, confidentiality, and availability of the application executable's file's contents and the data file's contents by preventing unauthorized writes and reads of a protected area of storage, where the application executable and associated data reside in this protected storage when not executing on a host PC (i.e. connected computer).

My previous patents, also referred to above, focused on an embedded hardware embodiment as a separate component of computer systems. The patent focused on the integration of the F+ Storage Firewall technology into a hardware storage controller.

The present invention covers storage controllers that are implemented as customizable software running on off-the-shelf processors and FPGAs rather than hardwired ASICs. This is made possible by the increasing performance and decreasing cost of microprocessors and FPGAs, reducing the cost, and improving the performance. This permits smarter more flexible storage devices to be used as components in what are currently expensive flash storage arrays (for data centers, cloud computing, etc.). There are a variety of other benefits, as well.

The present invention is described in terms of:

2 CSC designs illustrated in the drawing illustrate CSC architecture, with a significant architectural difference between them.

F+ Storage Firewall integration into these 2 CSC designs, or it may be more fair to say that the F+ Storage Firewall is extended into becoming a CSC.

To clarify language, where this document refers to 'storage controller', the often-used industry terms include disk controller, drive controller, and sometimes simply 'controller'. This document uses the phrase storage controller unless specifically referring to controllers for hard disk drives, in which case the phrase disk controller will be used.

A storage controller is presumed to be an integrated circuit chip or set of such chips, usually mounted on a circuit board, where this circuit board is bolted and wired onto (into) a storage device. This storage device may be a hard disk drive or may employ some other type of storage media such as a Flash (or flash) chips. This storage device itself is generally considered a component in a computer or other digital system of some type.

Where the term (Host) I/O Processor is found in the text, a common term in the storage industry is Host Bus Adapter, or HBA, or some other reference to the idea that the internal data bus is being bridged to the external storage and peripheral bus(ex PATA, SCSI, USB). Some of the recent prior art uses the North Bridge/South Bridge terms, where the North Bridge processor connects directly to the CPU core, and communicates with RAM on behalf of the CPU, etc., while the South Bridge processor communicates with the North Bridge and external storage device controllers such as Flash and hard disk drive (HDD) controllers.

In addition to the other glossaries and dictionaries mentioned, the IEEE dictionary is helpful.

The Authoritative Dictionary of IEEE Standards Terms. p. 438. doi:10.1109/IEEESTD.2000.322230. ISBN 0738126012.

Background terms, no change in their meanings is proposed by this disclosure

Binary blob, Blob	en.wikipedia.org/wiki/Binary_blob
Booting, Bootloader	en.wikipedia.org/wiki/Bootloader#Boot_loader
Disk controller	en.wikipedia.org/wiki/Disk_controller
Firmware	en.wikipedia.org/wiki/Firmware
FPGA—Field-programmable gate array	en.wikipedia.org/wiki/Field-programmable_gate_array
Hard disk drive	en.wikipedia.org/wiki/Hard_disk_drive
Hard disk drive interface	en.wikipedia.org/wiki/Hard_disk_drive_interface
Microcode	en.wikipedia.org/wiki/Microcode
Parallel ATA (IDE)	en.wikipedia.org/wiki/Integrated_Drive_Electronics
SAS (Serial attached SCSI)	en.wikipedia.org/wiki/Serial_Attached_SCSI
SCSI (Small Computer System Interface)	en.wikipedia.org/wiki/SCSI
SCSI command	en.wikipedia.org/wiki/SCSI_command
Serial ATA	en.wikipedia.org/wiki/Serial_ATA
ST-506	en.wikipedia.org/wiki/ST-506

Introduction to Storage Controllers

Storage Controller Basics

First, a basic definition of a storage controller: it is the interface and management mechanism, both hardware and software, that does the storage operations to storage media. Storage media can be some form of solid state memory chips (such as flash memory), or rotating magnetic platters (such as in a hard disk drive).

A storage controller allows a disk or flash drive to talk to a peripheral bus. Examples of a peripheral bus are PATA, SCSI, and USB, usually physically separate or external to the host circuit board(s).

The currently most common industry term for the component(s) that allows a computer host processor to talk to this peripheral bus is host adapter or host bus adapter (HBA).

The combination of a storage controller with storage media makes a storage device.

Storage access requests are received by a storage controller from a Host I/O Processor. The storage controller generally maintains a queue of access requests, where these requests (or commands) are formatted in according to which drive interface type is being supported; ex. AT commands for the SATA interface.

The storage controller is generally single threaded, managing a queue of storage access requests, translating in turn each received and queued access request into instructions for accessing the storage media, depending on the type of storage device.

The storage controller then translates and packages the reply, before passing it across the peripheral bus interface to the host interface.

Sometimes there may be another controller between a Host I/O Processor and a storage controller; this may be a storage array controller that supports the creation and management of RAID architectures. In some implementations, the disk array controller may be integrated into a Host I/O Processor, but is still performs different functions.

This applies to any size or shape or storage device. It is especially valuable for enterprise class storage devices, which often use the more intelligent Serial Attached SCSI (SAS) rather than the Serial ATA or Serial AT Attachment (SATA) more common on the desktop PC. These enterprise class storage devices are generally deployed in data centers, which are undergoing a disruptive evolution into cloud computing facilities.

Storage controllers also perform a variety of management functions, where "enterprise" class storage devices are expected to have more management functions than desktop class storage devices. In addition, enterprise class storage devices have greater performance, and other benefits to justify a correspondingly higher price point.

Recent History of the Development of Storage Controllers

This document omits the earlier history of storage controllers, from the 1960's programmable IOPS (Input/Output ProcessorS) through the SCSI, IDE, EIDE, and PATA technology generations as someone 'skilled in the art' is expected to already have a good grasp of this information.

The goals during this storage controller evolution have included:

Improving performance

Improving reliability and durability

Reducing cost, usually measured as per storage unit

Reducing power consumption

Simplifying storage protocols and physical connections, and reducing system complexity (leading to reduced costs)

Increasing the number of connected disks and other storage resources, which included increasing the distance of physical connections to storage

In 1999 a group of system, semiconductor and storage device suppliers recognized that the widely deployed IDE/PATA storage controller design would soon begin to limit system performance. These suppliers created the Serial ATA Working Group which set out to develop with the goal of a faster AT storage interface. The group released its Serial ATA 1.0 specification in August 2001. The Serial ATA Working Group later became The Serial ATA International Organization (SATA-IO, serialata.org)

The SATA standard replaced the older IDE and EIDE 40-conductor and 80-conductor flat ribbon cables used to connect disk drives to the motherboards of personal computers with thin and flexible point-to-point cables. SATA 1 transfers data at a higher rate (150 Megabytes per second) than PATA (typically 100 MB/second), while the current SATA 3 specifies 60 GB/s. Serial ATA storage devices became widely used during the second half of 2003, and have since become the most common interface type of storage device.

SATA continues to evolve, with security added by other groups, but newer storage interface technologies are being introduced such as PCIe, where the goal is higher performance through eliminating the latency caused by SATA ATA command processing. Some product manufacturers, such as Apple Computer, have eliminated or combined the storage controller functions with the I/O controller in current generation mobile devices (circa 2013).

Other storage controller (interface) types include (but are not limited to):

SCSI, which incorporated the drive controller earlier than IDE/PATA

SAS (a version of SCSI), used primarily for servers (often in data centers)

Fiber Channel, a networked controller technology used primarily in data centers

iSCSI (a networked version of SCSI) used primarily in data centers

PCIe, the new performance leader, for small systems of various types, eliminates the storage-specialized Host Interface **10**, relying on the PCIe controller to act as the Host Interface **10** to the storage device(s).

Current State of Storage Controllers, Including Problems

The current storage controller situation is in flux. Small systems (including desktops, small servers, laptops) and embedded devices have benefitted in the past from the ability to source common components such as standard disk drives. This has resulted in lower prices and faster revenue growth both for the disk drives and for the systems they are installed into.

Disk drive manufacturers were able to leverage higher manufacturing volumes into higher revenue growth, which enabled greater investment into research and development, which provided the world with improvements in disk drive (magnetic recording, read/write head architecture, and interface design) technologies which enabled disk drive density and cost to large track Moore's Law improvements in the density and cost of integrated circuits. So faster, denser, and cheaper computer processors and RAM were combined with faster, greater capacity, and cheaper disk drives, with standard interfaces, and commoditized

This path of evolutionary innovation would be expected to continue, but as the market situation is undergoing revolutionary changes, there are expected to be disruptive changes in deployed storage technologies. These changes include developments in solid state storage technologies which promote fragmentation of the market; some types of products see lower sales and other product types see faster growth than was expected. For example, the sales (and segment revenue) of desktop personal computers (PCs) have been lower, while mobile computers and related devices such as tablets have been much higher. The manufacturers of tablets do not see value in sourcing desktop PC storage components; instead they are generally either sourcing solid state storage devices (such as mSATA or PCIe) or are attaching solid state storage components directly to their circuit boards (such as in Apple laptops through early 2013).

For those system manufacturers still sourcing storage devices, whether solid state storage or disk drives, the devices are still block oriented, though the trend is larger block sizes. (512 byte blocks is the older standard, 4K byte 'sectors' is becoming the new standard, especially on Flash arrays. A larger sector size allows for higher capacity storage and larger size reads/writes on storage access, which supports higher performance storage architectures.) This block orientation made sense in the past with strictly standard interfaced hard disk drive storage devices, but makes much less sense when designing with solid state and hybrid storage components. One issue is that the operating systems and other technologies (such as data center oriented storage architectures) that interface to the storage components have their own market inertia.

Since there are different market-driven requirements for major market segments, these should be discussed separately. These include small systems (including mobile), and data center (including cloud) storage systems. Dividing the world into just these categories seems artificial as the real world is changing so rapidly, but lines have to be drawn somewhere, to get a handle on the changing requirements for storage controllers. In addition, data storage security requirements are rapidly changing and not currently being met.

Small system storage controllersSecurity

Compatibility with Data Center evolution

Small System Storage Controllers

As mentioned above, one of the trends for small systems, mobile devices, and large manufacturing volume embedded devices is to forgo use of standard storage devices if there are significant advantages to proprietary, proprietary, to:

Reduce chip count: and reduce required device real-estate and component (bill of material) costs

Reduce component (bill of material) costs

Improve performance through innovative storage interface schemes

SATA interfaced solid state storage devices should continue to be sourced by some manufacturers, but likely with extensions to the standard AT command set.

For small systems, in both the case of continued use of SATA (and other standard storage interfaces) and the use of proprietary storage interfaces, there will be a strong trend toward extensions to the older SATA interface standards in order to improve performance, reduce costs, etc. The security extension, Opal, is described below. Other extensions include Native Command Queuing. (Native Command Queuing (NCQ) specifies how storage devices can reorder read and write commands in order to optimize performance.) Some extensions will receive broad industry support and became part of the standard, while others will be proprietary to only one or some vendors' offerings.

Another trend is to replace SATA storage interface devices with PCIe, for better performance. Part of this performance increase is the reduced storage controller latency. The increased performance of flash solid state storage media has brought attention to the SATA performance penalty caused by the SATA storage controller's architecture. According to press accounts (such as electronicdesign.com/digital-ics/whats-difference-between-sata-and-nvme) forecasts are for 10% to 40% increase in design wins for PCIe for new products that formerly would have been expected to employ SATA storage devices. Manufacturers adopting PCIe include Apple Computer, which announced (www.theinquirer.net/inquirer/news/2274453/apple-looks-to-samsung-for-macbook-air-pciexpress-ssd) the use of PCIe in their next Macbook Air laptop.

Another emerging interface standard is NVMe (Non-Volatile Memory Express), which similar to PCIe, provides better performance than can be achieved with SATA 3 on flash storage devices.

All of these technologies and/interface types are evolving as a result of competitive pressures, to better meet current and expected market requirements.

Storage Device Security

Opal Introduction

A significant effort has been made to secure the contents of storage devices. One of the industry associations working on this is The Trusted Computing Group (TCG). Opal is the name of a specification related to self-encrypting drives, which has been developed by the TCG. Opal has some useful benefits, but also some significant problems, which are resolved or ameliorated by the integration of the F+ Storage Firewall and support system technology.

An Opal drive is a self-contained, stand-alone HDD or SSD that conforms to the TCG Opal standard. The drive is always encrypted but may or may not be locked. In addition to the standard components of a drive, an Opal drive will contain extra components such as an onboard cryptographic processor that perform all of the necessary encryption and decryption of data on the drive itself.

The Opal drive is a type of Self-Encrypting Drive (SED). It is not the only type of SED; there are other (proprietary) self-encrypting drive products (designs) in the market.

The primary Opal threat model use case is lost or theft of computers and drives. This is the same threat model as other encryption technologies such as software Full Disk Encryption (FDE) and is designed for protection of data at rest.

There are multiple versions of the Opal Standard. The current standard is Version 1.0. The TCG is currently discussing version 2.0 but that has not been ratified, nor are version 2.0 drives in production (as of the date this is written, 2012).

The Opal specification can be found on the web at http://www.trustedcomputinggroup.org/resources/tcg_storage_security_subsystem_
class_opal_version_100_revision_200
or at http://www.trustedcomputinggroup.org/files/static_page_files/B66DB236-1D09-3519-ADD7E75C7216311D/Opal_SSC_1.00_rev2.00-Final.pdf

Opal Goals

The TCG specification (Version 1.00, Revision 2.00) defines Opal goals as:

"

The Storage Workgroup specifications provide a comprehensive architecture for putting Storage Devices under policy control as determined by the trusted platform host, the capabilities of the Storage Device to conform with the policies of the trusted platform, and the lifecycle state of the Storage Device as a Trusted Peripheral.

"

Goals for Opal include

Drive level security

Simplicity

These enable (or are supported by):

Simplified management

Robust security

Compliance with national and state "safe harbor" laws

Scalable because no obstacles exist to massive arrays, etc.

Interoperable with existing systems, older disks, etc.

Integrated with management systems, etc.

Transparent (to the operating system, applications, etc.)

Opal Basics

From experience, the storage security community has learned that storage security requires both storage device locking and encryption of stored contents.

Locking only is easily hacked (ATA has locking)

Encryption only does not prevent access to data

Opal includes both Locking and Encryption, so attempts to find the right balance of features:

On power off, an Opal drive locks automatically.

On power on, an Opal drive remains LOCKED, until an authentication key (AK), a variety of password, is employed to unlock the drive. When unlocked, unfettered read and write access is permitted.

Opal Advantages

An Opal drive encrypts all data transferred to it transparently to the host. There is assumed to be no performance latency penalty because the encryption engine speed matches the storage interface's max speed, and because the encryption engine is a component of the storage controller.

All stored data is encrypted, so there is no need to determine which data to encrypt. And decryption and re-encryption not required when an Authentication Key (AK) is changed.

As Opal drives are added to a set (perhaps a disk array), encryption performance scales in a linear fashion (without added latency), because each Opal drive has its own encryption engine.

Standardization and the promise of interoperability haves led to support from multiple storage device vendors, multiple file server vendors, and multiple software vendors.

Opal Benefits include:

No Back Doors (assuming we trust the vendor)

No access to an Opal drive access without AK authentication, therefore resistant to unauthorized insider access

All stored data is always encrypted

Encryption cannot be turned off by a user

Encryption is not exposed outside of an Opal drive

Rapid "erase" of drive, such as by deleting or changing the data encryption key

On power off an Opal drive becomes locked, and is therefore presumed secure

Instant erase is provided by deleting the data encryption key (DEK), the symmetric key used for full disk encryption (FDE). This assumes the drive is sufficiently operable to submit the command; if the drive cannot accept the ERASE command, then it is likely not to permit storage access either, so presumably the data is safe.

The deployment of Opal drives is expected to reduce IT operating expenses and headaches because

There will no longer be a need (or cost) to overwrite or destroy drives

Drives can be repurposed securely

Warranty and expired lease returns won't expose data to unauthorized access

Use of verifiable encryption will provide "safe harbor" for data privacy laws (PCI, HIPAA, other)

Theft of data from loss of unencrypted drives is eliminated because all drives can be managed to activate encryption, therefore all stored data can always be encrypted

Opal Problems

Opal has a set of structural problems. The following is a list of Opal problems, perhaps not all:

(a) The current Opal specification does not address prohibiting unauthorized access after an Opal drive is unlocked, i.e. protecting the storage contents of an Opal drive while it is "in use",

(b) Nor does the current Opal specification address protecting accessed data while that data is "in use" in the host's memory.

(c) Opal also does not address protecting data while "in flight" between hosts.

(d) There is no way to wear level across drives in an array, and there is no way to move Opal encrypted data from a drive or between drives without decrypting the data. (On the other hand, there is no prevention of a second encryption of some or all data.)

(e) Opal Drives and the **S3** State: Sleep (Hibernation), Standby, Suspend to RAM (explained below)

(f) Opal Performance, Security (explained below)

(g) ERASE Command (explained below)

Opal Drives and the **S3** State: Sleep (Hibernation), Standby, Suspend to RAM

S3 is a power state, commonly known as Standby, Sleep (Hibernation), or Suspend to RAM. A system in an **S3** state consumes little power, but a previous operational state is available for fast re-activation. While in **S3** mode the CPU consumes no power, the RAM is in a slow refresh mode and the power supply is in a reduced power mode.

The problem with **S3** and Opal drives is that Opal drives Lock when they are not provided power, and it is difficult to see how to restart the operating system when the Opal drive is Locked and there is no built-in way to Unlock it. As of January 2012, the TOG (the Opal standards body) does not have a common and agreed-upon solution to the **S3** issue.

There are vendors such as WinMagic (Wave, McAfee, etc.) with drive management application software systems which may resolve the **S3** issue for some hosts in combination with some Opal drives.

Opal Performance, Security

It is intended for Opal drives to operate at the same speed as non-Opal drives with the same components. However it is difficult to determine whether runtime access latency is added by the cryptographic processing.

It is also difficult to determine whether a particular Opal drive's encryption engine is functioning as expected. Since there is no alternate path to sample encrypted data, and since the data encryption key can not be retrieved, it is impossible to verify whether the Opal encryption engine has properly encrypted the data.

ERASE Command

The intent of a drive erase operation is to insure that the storage contents cannot be retrieved, that the drive storage media is returned to a randomized state. When an Opal drive receives an ERASE command, the intent is for the data encryption key (DEK) to be is deleted or replaced, thus making it impossible to access previously encrypted data.

Trust in the Opal ERASE command requires one to have sufficient faith in the drive implementation, and in the strength of the encryption to withstand for some period of time novel developments in computation, mathematics, semiconductors, digital architectures, etc. Any attempt to forecast the effect of emerging and currently unknown technologies on the confidentiality of data whose persistent (non-volatile) storage media falls into the hands of hostile parties is, in my opinion, likely to end in the unfortunate situation of not knowing whether (or when) the data's confidentiality has been compromised.

Data Center Evolution and the Storage Device

The prevailing cloud, virtualization and storage architecture technologies are shaping the requirements for next-generation storage devices.

This is an overview of the big picture of the relationships among cloud computing, data center virtualization, software defined storage (SDS), storage systems, and storage devices, including flash storage devices. The evolution of cloud-related data center objectives will drive the requirements for next-generation storage device controllers, including features to support hypervisor integration, parallelism, security, data integrity, etc.

Introduction to Cloud Computing

Cloud computing is a much hyped marketing term for several competing distributed system architectures which attempt to run application software and deliver services based on these applications with a common set of benefits for customers. These customers each share a portion of the cloud computing infrastructure on an as-needed basis. The benefits include lower cost, greater uptime reliability, and the ability to rapidly scale application execution both up and down, as demand requires. By sharing the same infrastructure among many customers, many users, the cost of providing the cloud computing service is much lower than if each customer built their own data center with similar uptime reliability and high demand capabilities. These customers would otherwise have to make heavy investments in their own data center or not have access to this level of data center infrastructure. The 'magic' that makes cloud computing possible relies heavily on a set of technologies collectively referred to as virtualization.

Virtualization is a technology that supports the activity of multi-tenanted computers, a type of sharing of computer resources. The term virtualization refers to the creation of a virtual machine that acts like a real computer with an operating system. Software executed on these virtual machines is encapsulated from the underlying operating environment. The host computer is the actual hardware on which the virtualization takes place. The virtual computer is sometimes referred to as a guest computer. The software that creates and manages a virtual machine on the host computer is called a hypervisor.

Cloud computing is built up from the use of virtualization on "server farms", with supporting technologies including distributed fault-tolerant storage resources such as storage arrays, RAID, Network Attached Storage (NAS), etc.

Evolution of the One-Size-Fits-all Cloud

Currently, the one-size-fits-all cloud is being transformed by market pressures.

The current cloud approach is for a cloud service provider to force all applications to fit a common set of virtualization technologies and underlying infrastructure.

Cloud infrastructure, especially infrastructure-as-a-service, will increasingly rely on more flexible storage architectures. There are attempts being made to develop more flexible technology for cloud computing infrastructure. This broad category of more flexible storage architectures is sometimes referred to as 'software defined-storage' (SDS). This is an emerging approach to better utilize storage resources.

This more flexible approach to data center storage architecture will initially improve application performance, but will later be part of the new cloud technology foundation, which will enable next-generation applications, new revenue streams and profitability for the cloud providers and their customers.

Storage Architecture Evolution, Infrastructure-as-a-Service, Software Defined Storage

Software defined storage (SDS) can be currently characterized as an attempt to improve performance from flash based storage devices yet still provide management and other functionality.

Current software-defined-storage (SDS) thinking about the ways to apply the faster Flash storage devices:

Take away management latency from the read/write path on storage devices in order to free up performance for applications that both benefit from the fastest possible performance and can handle 'raw' storage access. These include database, ERP, etc.

Then add a comprehensive abstraction layer for storage management, by moving management functionality from the storage device to a new storage appliance layer. This supports data center & cloud computing trends, plus enables the combination of heterogeneous storage such as block mode, object storage, etc., into a single storage set, even from multiple vendors. This means that the enterprise storage device will increasingly be seen as a commodity component, and that the storage appliance layer is available to use as a foundation for infrastructure-as-a-service.

Security

The other side of the problem being solved is security. It is a premise, a foundational belief, that the integrity of data is only as trustworthy as each and every one of the digital system components that have contact with this data. Cloud computing relies on distributed systems technologies where even the location of their data is hidden from most of the owners and users, and where the there must be 100% assurance that data being used (such as in calculations and decisions) has not been tampered with, yet there can not today be 100% assurance that none of the relevant computing and network components have been compromised. This is a significant and costly problem for both customers and cloud vendors: customers because they will keep certain IT operations in their private data processing facilities where they have a greater assurance of security, and cloud vendors because they lose the additional revenue they would have achieved were they able to prove a verifiable level of security including data integrity.

Requirements for Next-Generation Storage Controllers

Requirements for next-generation flash storage device controllers include features such as parallelism, hypervisor integration, etc. based on a logic trail from evolving cloud data center goals.

The commercial success of the emerging next-generation storage controller will be based on characteristics and abilities of the storage controllers' hardware base, as well as evolving requirements of the larger storage and device architecture. Some of these are:

Faster general-purpose (COTS) processors becoming cheaper and faster, suitable for use as storage controller processors,

Compliance with new and evolving standards for security such as from the Trusted Computing Group (TCG), the US Department of Commerce National Institute of Standards and Technology (NIST), and other government agencies and industry groups where as these evolve existing infrastructure has to evolve accordingly.

Data retention requirements and the costly risk of losing access to archived data as server and data center storage access technologies evolve away from that prevailing when a storage device was manufactured.

- Rapidly evolving storage access requirements, including compatibility with data center trends such as cloud computing, data center virtualization, and software defined storage (SDS).

The Present Invention: Customizable Storage Controllers (CSCs)

The CSC is a set of improvements to the storage controllers on storage devices. These storage devices can be hard disk drives (HDDs), flash drives (UFDs or FDs), and solid state drives (SSDs). These CSC improved storage devices can be used in all of the same use cases (situations and locations) where the unimproved storage devices can be used, but with better performance, greater reliability, etc. CSC improved storage devices can also be used in new use cases, such as new data center storage architectures.

This disclosure focuses attention to SATA storage devices, but the principles and invention can be applied to any storage interface, including networked storage interfaces, including but not limited to Fiber Channel (a network connected storage interface deployed in some data centers).

The earlier parts of the disclosure showed how a F+ Storage Firewall can sit between a host interface and a storage controller, and how the F+ Storage Firewall technology can be integrated seamlessly into a storage controller, but consider: if the storage controller is just a set of one or more software executables, then the F+ Storage Firewall can manage and protect these software executables just as it would any of the contents of the protected storage. And the F+ Storage Firewall support system can be used to manage, maintain, upgrade, monitor, etc. the storage controller executable software. Thus, the CSC can be viewed as an evolutionary step for storage controllers, or as an embodiment of the F+ Storage Firewall.

The most significant differences of the CSC from the Prior Art storage controllers are:

- Current data center storage architectures, storage array controllers, and operating systems are forced to conform to the capabilities of the interface type of the storage devices being used. This means that the technologies and operating environment that surround and rely on the storage devices have to be designed around the weaknesses of the current generation of storage controllers
- A storage architecture that takes advantage of the CSC can adjust the operating configuration and even swap-out the CSC software, in order to make the CSC controlled storage device conform to the requirements of the storage architecture and/or host operating environment. This also means that some or much of the storage interface stack and I/O processing that otherwise would have to be done by the storage architecture can be done by the CSC controlled storage device. And, if the CSC is permitted to communicate with it's peers, then additional functionality is enabled, such as greater parallelism, redundancy, fault tolerance, and even application level functionality such as RAID-like fault tolerance and recovery.
- In practical terms, the CSC approach requires the integration of security technologies such as are provided by the F+ Storage Firewall and support system, in order to protect the storage controller executables and therefore to protect the integrity of storage controller operation.

This invention proposes a third approach, where the controller software on the storage device can be customized on an as-needed basis, as frequently as needed, with greater intelligence on the storage devices. This would enable:

- larger sets of storage devices to be clustered, with a reduced runtime burden on the storage appliance layer
- ad-hoc sets of storage devices, supporting the flexibility goals of infrastructure-as-a-service
- future inventions of novel storage architectures
- greater security with the integrated F+ Storage Firewall with support system (executable whitelist access control, etc.)

This invention further proposes that these improved storage devices can have two controller processors, where one of them is a fast and secure FPGA processor, for fast path storage access, while the other is a single or multi-core processor for storage management, volume management, RAID, snapshots, etc. The F+ Storage Firewall and support system has the right features to manage, monitor, maintain, and protect the customizable storage controller's executable software and configurations, as well as to protect the contents of the storage device.

This is an important area because storage access interfaces are and will be an area of rapid change, and this innovation makes it possible to support not only the current and emerging storage access interfaces (connectors and protocols), but also as-yet undefined storage access interfaces, so long as the physical connectors are compatible.

There are many significant changes being made to the virtualization and storage architecture technologies in use. All of these can derive performance and reliability benefits from customizable storage controllers CSCs integrated into customizable storage devices.

Software Defined Storage and the Storage Controller

The movement towards Software Defined Storage (SDS) in the data center is about

- maximizing data center IT investment by providing for 'pooling' of storage resources, assigning them as needed
- automating the provisioning of storage infrastructure and corresponding applications
- improving IT operations (incl. control) through centralized management for 'health', risk, efficiency, and compliance.

There is a corresponding movement towards SDS for smaller servers and on the desktop, as the these computing environments evolve to better coordinate with the rapidly evolving data center and cloud computing environments.

The CSC fits into the SDS scheme by providing a mechanism for SDS centralized management systems to set corresponding and compliant CSC software and configurations into the managed storage devices.

The CSC software can support data center SDS virtualization in several ways. One of these is with multiple defined caches corresponding to the virtual environments, in order to enables a reduction in latency caused by stepping on cache contents, improving performance by reducing the number of times there must be access operations to retrieve frequently accessed data. The specific CSC storage controller adaptations for particular deployment scenarios will include a variety of features necessary for high availability and performance in these deployments.

The update and other support mechanisms already described for the F+ Storage Firewall will continue to be applied to the CSC, leveraging the same distributed management support system.

The customizable storage controller with integrated F+ Storage Firewall (the CSC approach) supports these SDS/Infrastructure-as-a-Service goals by

- Enabling flexibility in storage architecture and driving cloud virtualization into the storage device, and by

- Securing each component, each storage device, to provide assurance of data integrity as well as system integrity, data confidentiality, and reduced downtime through protection of storage device availability.

The CSC approach will transform SDS as storage architects make use of the new flexibility in storage interface design.

The CSC approach transforms the cloud by improving the ability of cloud architects to define the functionality of storage architectures through to the storage interfaces, data formats, and other storage device functionality on an as-needed basis. Cloud computing storage devices can be as provision-able and modifiable as other aspects of the virtualization foundations of the cloud, with virtualization environments directly supported by compliant (customizable) storage devices.

The present invention will be described referring to FIGS. **1**, **5**, **7**B, and new FIGS. **8**A, **8**B, **9**A, and **9**B.

FIG. **8**A Customizable Storage Controller, Single Processor

FIG. **8**A puts the single processor CSC invention in context with a Host computer or other digital system. FIG. **8**A contains an illustration of a Customizable Storage Controller with a Single Processor. This CSC is labeled Single Processor CSC **310**.

The customizable storage controller Single Processor CSC **310** is an improved version of SSD Controller w/Storage Firewall Components S**2** of FIG. **7**B.

The Single Processor CSC **310** contains a possibly multi-core SC Processor **314**, enough SC RAM **316** to hold the storage controller executable software plus any other desired contents, and the SC Program Store **318**, which may employ a portion of the (FIG. **5**) Protected Storage **14**, and/or may employ an independent storage element (storage media).

The Single Processor CSC **310** is communicatively coupled over interface **308** to I/O Processor **306**, which is a component of Host **300**, which is equivalent to Host **204** on FIG. **5**.

The Single Processor CSC **310** is integrated with the Storage Media **312** in a manner similar to the integration of SSD Controller w/Storage Firewall Components S**2** of FIG. **7**B with the one or more of elements S**7**, S**8**, and S**9** of FIG. **7**B.

FIG. **8**B Customizable Storage Controller, Single Processor, Internals

FIG. **8**B illustrates how the components of a simple CSC with single processor are organized.

The Single Processor CSC **310** has the internal block diagram and data path illustrated on FIG. **8**B.

The Single Processor CSC **310** has a comparatively simple design. It is basically a simple computer that processes storage access requests, storage device management requests, and other management functionality.

The storage access data path and storage device request queue are not shown.

Single Processor CSC **310** is communicatively coupled to the Host over interface **308**, and is integrated with the Storage Media **312** over storage media interface **344**, which is equivalent to one or more of elements S**7**, S**8**, and S**9** of FIG. **7**B.

On the Single Processor CSC **310**, the SC Processor **314** connects to SC RAM **316** over internal bus **342**, and connects to SC Program Store **318** over internal bus **344**.

The SC Processor **314** may often have several different software executables competing for attention; these will be processing data access requests, managing the request queue (not shown), modifying the contents of the SC Program Store **318**, etc.

There is also an internal bus connection **346** between SC RAM **316** and SC Program Store **318**, for loading software, runtime swap space, etc.

FIG. **9**A Customizable Storage Controller with Security Coprocessor

FIG. **9**A puts the improved CSC invention in context with a Host computer or other digital system. FIG. **9**A contains an illustration of a Customizable SC with Security Coprocessor. This is labeled CSC with Security Co-Proc **320**.

Continuing with FIG. **9**A, the Customizable SC with Security Coprocessor **320** is an improved version of SSD Controller w/Storage Firewall Components S**2** of FIG. **7**B.

The CSC with Security Co-Proc **320** contains both a possibly multi-core SC Processor **326**, and a Security Coprocessor **322**, enough SC RAM **328** to hold the storage controller executable software plus any other desired contents, and the SC Program Store **320**, which may employ a portion of the (FIG. **5**) Protected Storage **14**, and/or may employ an independent storage element (storage media), and Locked Firmware **324** which may employ a portion of the (FIG. **5**) Protected Storage **14**, and/or may employ an independent storage element (storage media).

The distinction between the SC Program Store **320** and the Locked Firmware **324** relates to how well protected the contents are to change, which is related to the difficulty of making authorized changes. There is a further implication, but not a requirement, that the Locked Firmware **324** contains functionality that is less likely to change probably because it is composed of lower level atomic operations, as well as a trusted factory approved version of the storage controller software, enabling recovery if there is a problem. In general, but not exclusively, the Security Coprocessor **322** will load its executable software from the Locked Firmware **324**, while the SC Processor **326** will load its executable software from the SC Program Store **320**.

The CSC with Security Co-Proc **320** is communicatively coupled over interface **308** to I/O Processor **306**, which is a component of Host **300**, which is equivalent to Host **204** on FIG. **5**.

The CSC with Security Co-Proc **320** is integrated with the Storage Media **312** in a manner similar to the integration of SSD Controller w/Storage Firewall Components S**2** of FIG. **7**B with one or more of the elements S**7**, S**8**, and S**9** of FIG. **7**B.

FIG. **9**B Customizable Storage Controller with Security Coprocessor, Internals

Focusing on the CSC with Security Co-Proc **320**, the distinction between the two processors includes that the Security Coprocessor **322** protects the customizable storage controller's operations in several ways, while the SC Processor **326** enables and supports the flexibility and extensibility of the customizable storage controller.

One embodiment has the Security Coprocessor **322** performing the storage access operations, including access control aspects of the storage access processes, while the SC Processor **326** performs management functions. The Security Coprocessor **322** may be implemented by an FPGA, perhaps a non-volatile FPGA, for fast processing of storage access requests, as well as other security functions as called by the storage controller software executing on the SC Processor **326**.

In the illustrated embodiment, the Security Coprocessor **322** can load software from the Locked Firmware **324** over internal bus **350**, but can not write directly to the Locked Firmware **324**.

In the illustrated embodiment, the SC Processor **326** can load software into the Locked Firmware **324** over internal bus **356**, but can not read directly from the Locked Firmware **324**.

The Security Coprocessor **322** is communicatively coupled with the SC Processor **326** over internal bus **354**, and is communicatively coupled with SC Program Store **330** over internal bus **352**. Not shown, but possible in some embodiments, is a memory segment of SC RAM **328** shared between Security Coprocessor **322** and SC Processor **326** for various purposes such as collaborating on operations.

The SC Processor **326** is communicatively coupled with SC RAM **328** over internal bus **358**, and is communicatively coupled with SC Program Store **330** over internal bus **360**. SC RAM **328** is communicatively coupled with SC Program Store **330** over internal bus **362**.

The SC Processor **326** may often have several different software executables competing for attention; including management of the request queue (not shown), modifying the contents of the SC Program Store **330**, etc.

Customizable Storage Controllers (CSCs)

The goal of a Customizable Storage Controller (CSC) is to provide a mechanism for CSC functionality to be modified by manufacturers, integrators, IT data centers (end-user organizations), and others to better meet performance and other metrics.

Customizable storage controllers (CSCs) are an approach to the design of intelligent storage controllers that will provide better performance in the emerging software defined storage (SDS) and virtualization environments.

The controllers are designed with features to introduce parallelism, etc. Also implementing the virtualization entity in controller hides the physical storage complexity. A CSC can therefore be thought of as a software defined storage device controller, a replacement for the more rigidly defined and programmed storage controller approach that has been used up to now.

While this description focuses attention on SATA, other storage interface types such as SCSI, SAS, Fiber Channel, etc. can be improved in the same way.

The CSC is a new approach to storage device controller design; post-deployment, CSCs can adapt to current requirements; either new (changed) CSC software, changes to configuration, even new control commands, extensions to the AT command set, even replacements for the AT command set with an entirely different storage access architecture and interface. And if access to the current data on the storage media must be preserved, so long as the new storage controller software supports the current storage media format, there should not be any obstacle to the continued use of this storage device.

CSC Architecture

An improved (more secure) CSC can be designed with a security co-processor and locked firmware, again both with or without the integration of the F+ Storage Firewall.

These designs can be implemented with standard parts such as microprocessors and/or FPGAs (Field Programmable Gate Arrays), RAM (Random Access Memory), and some version of nonvolatile memory as a program store.

There are several (i.e. 3) approaches to customizable storage controllers described, each with a corresponding integrated F+ Storage Firewall embodiment. It should be possible to make claims to the several approaches to CSCs as well as to the F+ Storage Firewall integrated into them.

The first is a simple architecture incorporating a storage controller processor, some sort of RAM, and some mechanism to store storage controller software, which may be separate from the storage resource being managed, or may be integrated into the storage resource being managed.

The second adds a security coprocessor and trusted “locked firmware”. This can be implemented in several ways, including through the use of a second microprocessor and an EEPROM containing its software. A better approach is to use an FPGA where the processor and its software are combined as the cell (gate) contents.

The key difference between the two versions is the level of security that can be provided by providing a secure startup for the security coprocessor, which itself then provides a secure startup and a hardware ‘root of trust’ for the storage controller processor.

Another way to design the CSC is using an FPGA or an ASIC for ‘storage primitives’, i.e. low-level storage operations, to improve performance. In this scheme, the CSC software is more easily changed (updated, augmented, etc.) while the set of storage primitives is not as easily changed or is not change-able. This is the preferred embodiment of the CSC. In this scheme, there is still a secure co-processor, built into the FPGA or ASIC.

In all of these versions (embodiments), the F+ Storage Firewall can be integrated for authentication, etc., to prevent unauthorized access to protected storage contents.

Assume security co-processor can be either a 2nd microprocessor, or separate cores on the same processor, or an FPGA. The preferred embodiment is the use an FPGA as security co-processor combined with a multi-core CSC processor for management and better integration into higher level (external to storage device) storage architectures. As these external storage architectures will continue to evolve, the CSC software can evolve in compatible ways, with changes to the higher performance FPGA software (firmware, IP cores) less frequently. It is possible, and a preferred embodiment, to extend the parallelism of the CSC into the FPGA.

A CSC does not necessarily require replaceable base storage controller software to operate—there are possible ASIC embodiments of the base controller software, but the use of replaceable software simplifies the architecture and deployment. [It is assumed by this description that the operative configuration of an FPGA is a type of software.] An ASIC CSC would be customizable through changes to its configuration, and if permitted, through the addition of plug-in or add-on modules to be executed on an associated coprocessor.

There are several possible data paths and the use of one in an illustration does not prevent the use of others in the same CSC.

The existing manner of designing SATA and other controllers is generally to single thread storage access operations

The difference of CSC functionality and behavior enabled by the integrated F+ Storage Firewall technology are documented earlier in this disclosure. The F+ Storage Firewall Transaction Processor **20** (as seen in FIG. **5**) and other F+ Storage Firewall components are interwoven into the CSC software; it's also fair to say that the CSC software is built on top of the F+ Storage Firewall and support system.

If there is a Security Co-Processor

If present, the use of the Security Coprocessor **322** changes the startup and normal operation of the CSC.

A preferred embodiment of the Security Coprocessor **322** is as an FPGA (Field Programmable Gate Array), where the Locked Firmware **324** can be stored in a persistent store such as Flash, or the FPGA can be a so-called persistent or nonvolatile FPGA. In either case, the FPGA (Security

Coprocessor **322**) must be loaded with software (firmware) and active before anything else will happen.

The Security Coprocessor **322** probes and initializes hardware resources and the contents of the Locked Firmware **324** during boot. This is about the same as the startup process for the F+ Storage Firewall device **12** illustrated in FIG. **1**.

Assuming the Security Coprocessor **322** is implemented by an FPGA, the F+ Storage Firewall state table and transaction processing discussed earlier in this disclosure can be implemented in the FPGA, for improved performance and some resistance to tampering.

Once the Security Coprocessor **322** has verified itself and necessary resources, it verifies the CSC software stored in Program Store **320**, then loads this software into the CSC processor, SC Processor **326**, after which normal operation can start.

Performance

The possible additional latency created by CSC management and other enhanced functionality is avoided through use of an FPGA or ASIC to provide the fastest possible data path, while enabling the CSC processor to provide the flexibility and other benefits of the CSC.

CSCs are a New Approach

This is a new approach, in that post-deployment, CSCs can adapt to current requirements; either new (changed) CSC software, changes to configuration, even new control commands, extensions to the AT command set. Also the CSCs can provide benefits such as high availability, scalability, load balancing and failover.

A CSC does not necessarily require replaceable software to operate—there are possible ASIC embodiments, but the use of software simplifies the architecture. [It is assumed by this description that the operative configuration of an FPGA is a type of software.] An ASIC CSC would be customizable through changes to its configuration, and if permitted, through the addition of plug-in or add-on modules.

A Customizable Storage Controller (CSC) can therefore be thought of as a software defined storage device controller, a replacement for the more rigidly defined and programmed storage controller approach that has been used up to now.

The differences from the current storage controllers include that the CSC software will need to be protected from unauthorized modification and that it provides an excellent place to add additional storage management functionality, permitting a better fit for the evolving data center Software Defined Storage and other virtualization trends.

The CSC type of storage controller is a good place to integrate the F+ Storage Firewall storage protection technology, fitting the needs of the CSC as well as protecting stored data from unauthorized access. Another way of looking at this is that the F+ Storage Firewall concept can be gently expanded to support the management and support of CSC software executables, or slightly more expanded to become the CSC, with the added benefit of runtime security. This is the approach taken by this invention, that the F+ Storage Firewall assumes the role of storage controller, with all of the management, maintenance, upgradeability, and other benefits offered by the F+ Storage Firewall architecture.

The best way to build this F+ Storage Firewall/CSC is to put into a fast processor (presumably an FPGA) functions that benefit from fast performance, and/or that don't change often, and/or that will benefit from parallelism and other FPGA features, and/or other reasons, while operations that change more often or that are of more of an executive management flavor run on a traditional processor. It is possible that this traditional processor is a multicore processor or even a set of traditional processors.

While this description focuses attention on SATA, other storage interface types such as SCSI, SAS, Fiber Channel, etc. can be improved in the same way.

The security co-processor can be either a 2nd microprocessor, or separate cores on the same processor, or an FPGA. The preferred embodiment is the use an FPGA as security co-processor combined with a multi-core CSC processor for management and better integration into higher level (external to storage device) storage architectures. As these external storage architectures will continue to evolve, the CSC software can evolve in compatible ways, with changes to the higher performance FPGA software (firmware, IP cores) less frequently. It is possible, and a preferred embodiment, to extend the parallelism of the CSC into the FPGA.

CSC Operations

The normal operation of the Customizable SC with Security Coprocessor **320** (of FIG. **8**) relies on interactions between the Security Coprocessor **322** and SC Processor **326**.

There are several ways to organize these interactions. These include preferred embodiments:

SC Processor **326** is involved in all operations, making use of portions of the Security Coprocessor **322** FPGA with subroutine calls to handle some of the time-consuming aspects of the operations. The SC Processor **326** "calls" to Security Coprocessor **322** (FPGA) software in ways documented in the Reference material. This leverages the greater parallelism and performance of an FPGA, as well as the greater presumed trustworthiness of the locked software. This sort of architecture has been applied to other types of digital systems such as a stock trading system. (see References). There are no existing known storage controllers with this architecture.

The Security Coprocessor **322** (FPGA) handles routine storage access operations, with the SC Processor **326** handling less common operations, and providing a way to upgrade the locked software executing within the Security Coprocessor **322**. There are no existing known storage controllers with this architecture.

There need not be any difference in the physical chip layout between these, assuming sufficient capacity, and therefore the Customizable SC with Security Coprocessor **320** can be converted from one to the other as needed.

Although the present invention has been described above in terms of specific exemplary embodiments, it will be understood by those skilled in the art that the disclosures herein are intended to be merely illustrative and not limiting in any way. Accordingly, the following claims are to be interpreted as covering all applications, modifications, variations and extensions as fall within the true spirit of the invention.

What is claimed is:

1. A data storage apparatus, comprising:

a host interface for coupling said storage apparatus to a host computer and/or other digital system;

a protected storage component;

a customizable storage controller operatively associated with said protected storage component;

a storage controller program store; and

a storage firewall adapted to communicatively couple said protected storage component and customizable storage controller with said host interface, said storage firewall being integrated with said customizable storage controller such that the combination thereof is operative to provide

software authentication including application registration,

runtime authentication of software identity and permission to execute,

authentication & authorization in the execution of executable software, and

examination, verification, and authentication of all storage access requests;

a security coprocessor operatively coupled to said storage controller program store, and coupling said storage controller processor to the host computer or other digital system via the storage firewall and host interface; and

locked firmware connected to said storage controller processor and said coprocessor, and operative to contain trusted source storage controller executable software for enabling controller recovery in the event of a problem and/or providing an addition level of protection against unauthorized change of the firmware.

2. A data storage system comprising:

data storage apparatus;

an integrated customizable storage controller processor;

a storage controller program store;

a host interface for coupling said storage apparatus to a host computer;

protected storage media;

a security coprocessor operatively coupled to said storage controller program store, and coupling said storage controller processor to the host computer or other digital system via a storage firewall and the host interface; and

locked firmware connected to said storage controller processor and said coprocessor, and operative to contain trusted source storage controller executable software for enabling controller recovery in the event of a problem and/or providing an addition level of protection against unauthorized change of the firmware;

wherein the customizable storage controller processor and storage firewall are adapted to communicatively couple said protected storage media and said host interface, said integrated customizable storage controller processor and storage firewall being operative to provide storage controller functionality including

protected storage media access operations including read and write operations, and/or

updates, upgrades, and reconfiguration of storage apparatus while in operation in parallel with the storage media access operations, and/or

protected storage device management, and/or

storage media management, and/or

storage device monitoring, and/or

updates, upgrades, and reconfiguration of said host interface, and/or

data encryption/decryption, and/or

authentication of updates, upgrades, and reconfiguration data, and/or

authentication and registration of software, and/or

runtime software authentication and/or authorization and/or permission to execute in the execution of software, and

examination, verification, and authentication of all storage access requests.

3. A data storage apparatus as recited in claim **1** wherein said customizable storage controller includes

a storage controller module having

a storage controller processor;

storage controller RAM; and

storage controller program store; and

wherein said protected storage component includes storage media controlled and accessed by the storage controller processor.

4. A customizable storage controller for use in a data storage apparatus including a protected storage component, a host interface for coupling the data storage apparatus to a host computer and/or other digital system, and a storage firewall adapted to communicatively couple the protected storage component and said host interface, said customizable storage controller comprising:

a storage controller processor coupled to the host computer and/or other digital system via the storage firewall and host interface, said storage controller being adapted to process storage access requests, storage device management requests, and other storage management functionalities;

a storage controller program store communicatively connected to said storage controller processor via an internal bus;

storage controller RAM operatively connected between said storage controller processor and said program store, said RAM being operative to fetch executable software from said program store and hold it for execution by said storage controller processor;

a security coprocessor operatively coupled to said storage controller program store, and coupling said storage controller processor to the host computer or other digital system via the storage firewall and host interface; and

locked firmware connected to said storage controller processor and said coprocessor, and operative to contain trusted source storage controller executable software for enabling controller recovery in the event of a problem and/or providing an addition level of protection against unauthorized change of the firmware.

5. A customizable storage controller as recited in claim **4** wherein a principal function of said security coprocessor is to protect the customizable controller's operations, and the principal function of the storage controller processor is to enable and support the flexibility and extensibility of the customizable storage controller.

6. A customizable storage controller as recited in claim **4** wherein said security coprocessor is implemented with an FPGA for enabling fast processing of storage access requests as well as other security functions as called by the storage controller software executing on the storage controller processor.

7. A customizable storage controller as recited in claim **4** wherein said security coprocessor can load software from the locked firmware over an internal bus, but cannot write directly to the locked firmware.

8. A customizable storage controller as recited in claim **4** wherein any needed executable and authenticatable software can be enabled and input from a trusted external digital system into the storage processor program store, protected storage component and/or locked firmware.

9. A customizable storage controller as recited in claim **4** wherein controller software on any storage device can be customized on an as-needed basis to enhance its intelligence capability, to enable larger sets of storage devices to be clustered with reduced runtime burden on the storage appliance layer, and to enable ad-hoc sets of storage devices supporting the flexibility goals of infrastructure-as-a-service storage architectures with greater security.

10. A customizable storage controller as recited in claim **4** wherein the storage firewall of the data storage apparatus is an F+Storage Firewall integrated with said customizable storage controller to support SDS/Infrastructure-as-a-Service goals by enabling flexibility in storage architecture and driving cloud virtualization into the storage device, and securing each associated storage device to provide assurance of data integrity as well as system integrity, data confidentiality, and reduced downtime through protection of storage device availability.

* * * * *